NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND ANALYSIS OF A
NETWORK INTERFACE FOR THE
MULTI-LINGUAL DATABASE SYSTEM

by

Clemon R. Wortherly

December 1985

Thesis Advisor: D. K. Hsiao

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: aval Postgraduate School; 6b. OFFICE SYMBOL (If applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): onterey, CA 93943-5100	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5100
NAME OF FUNDING / SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

TITLE (Include Security Classification)
IE DESIGN AND ANALYSIS OF A NETWORK INTERFACE FOR THE MULTI-LINGUAL
TABASE SYSTEM (UNCLASSIFIED)

PERSONAL AUTHOR(S)
emon R. Wortherly

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
ster's Thesis	FROM _____ TO _____	19 December 1985	127

SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multi-lingual Database System (MLDS), Multi-backend Database System (MBDS), Attribute-based Data Model, Attribute-based Data Language (Cont)

ABSTRACT (Continue on reverse if necessary and identify by block number)

aditionally, the design and implementation of a conventional database stem begins with the choice of a data model followed by the specifications a model-based data language. Thus, the database system is restricted to single data model and a specific data language. An alternative to this aditional approach to database-system development is the multi-lingual tabase system (MLDS). This alternative approach affords the user the ility to access and manage a large collection of databases, via several ta models and their corresponding data languages, without the aforemen-oned restriction.

this thesis, we present a methodology for supporting network (CODASYL) tabase management on the MLDS. Specifically, we design an interface ich translates CODASYL-DML statements into ABDL requests. We describe e data structures, the control mechanisms, and the functions/procedures cessary to implement such a system.

DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified

NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
rof. David K. Hsiao	408-646-2253	52Hq

ORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Block # 18 (Continued)

(ABDL), CODASYL Data Model, Network Database Translation

Approved for Public Release, Distribution Unlimited.

The Design and Analysis of a Network Interface for the Multi-Lingual Database System

by

Clemon R. Wortherly
Lieutenant, United States Navy
B.S., University of Texas, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1985

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases, via several data models and their corresponding data languages, without the aforementioned restriction.

In this thesis, we present a methodology for supporting network (CODASYL) database management on the MLDS. Specifically, we design an interface which translates CODASYL-DML statements into ABDL requests. We describe the data structures, the control mechanisms, and the functions/ procedures necessary to implement such a system.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

During the past two decades, the design and implementation of database systems has followed a rather predictable path. The sequence of events in the typical approach has been to decide on a data model, specify a model-based language, and ultimately, develop a system for controlling and executing the transactions written in the data language. This approach to database system development has resulted in an abundance of homogeneous database systems each of which restricts the user to a single data model and a specific model-based data manipulation language. Some examples of systems developed using this approach include IBM's Information Management System (IMS) which supports only the hierarchical data model and Data Language I (DL/I), Sperry Univac's DMS-1100 which supports just the network data model and the CODASYL Data Manipulation Language, and IBM's SQL/Data System which supports solely the relational data model and the Structured English Query Language (SQL).

An unconventional approach to the problem of database management system development, referred to as the Multilingual Database System (MLDS) [Ref. 1], eliminates the restrictions mentioned above. The MLDS would give the user

the ability to access and manage a large collection of databases, using several data models and their corresponding data languages. The design goal of the MLDS project is the development of a system that is accessible via a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL interface, and an entity-relationship/Daplex interface. Such a system would function as if it were a heterogeneous collection of database systems instead of a single model, single language system.

Some of the advantages of a MLDS are the reuseability of database transactions developed on a conventional system, economy and effectiveness of hardware upgrades (since we now upgrade just one system instead of a number of different systems), and its ability to support a variety of databases built around any of the well-known data models. Thus, there is ample motivation for developing such a system as the MLDS.

B. THE SYSTEM ORGANIZATION

In order to realize the above capabilities, the MLDS must be supported by an underlying database system that is both fast, efficient, and effective. If these criterion are not met, then the interfaces being developed for the MLDS may be rendered useless. Hence, it is important that the kernel data model and kernel data language (the underlying model and language for the system) be supported by a high-performance and high-capacity database system. Currently,

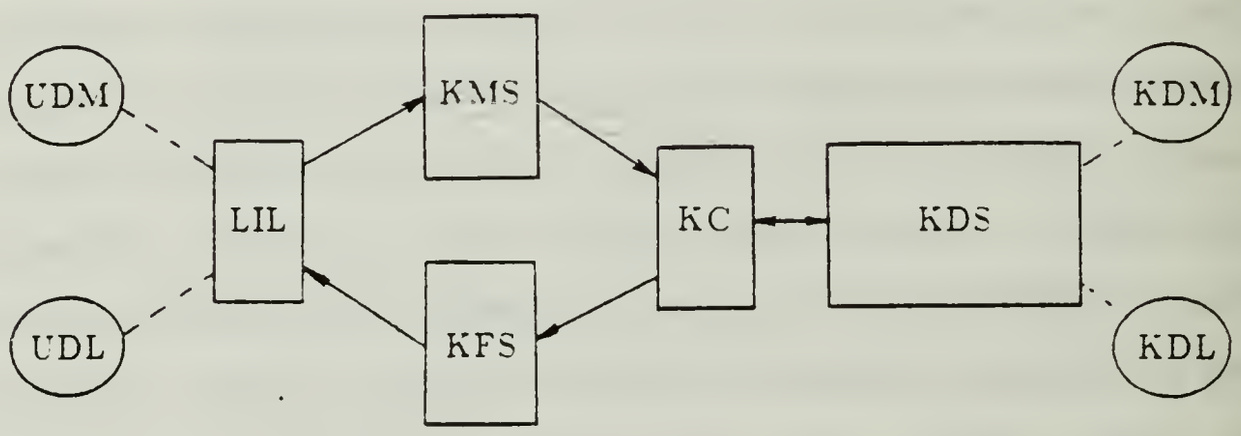
the attribute-based data model and attribute-based data language are the underlying model and language of a system which is referred to as the Multi-backend Database System (MBDS). In this section, we provide an overview of both the MLDS and the MBDS to enhance the readers understanding of the entire Multi-Lingual Database System.

1. The Multi-Lingual Database System

Figure 1 illustrates the complete structure of the multi-lingual database system. The user interacts with the system through the language interface layer (LIL), using a chosen user data model (UDM) to issue transactions written in a corresponding model-based user data language (UDL). The LIL routes the user transactions to the kernel mapping system (KMS). The KMS then performs one of two possible tasks. It either transforms a UDM-based database definition to an equivalent database definition based on the kernel data model (KDM); or, when the user specifies that a UDL transaction is to be executed, it translates the UDL transaction into an equivalent transaction in the kernel data language (KDL).

The first task is performed in the following way. The KMS forwards the KDM data definition to the kernel controller (KC). The KC then sends the KDM database definition to the kernel database system (KDS). When the KDS is finished with processing the KDM database definition, it informs the KC. The KC then notifies the user, via the LIL, that the database definition has been processed and

that the loading of the database records may begin. The second task is performed in a similar fashion. The KMS sends the transactions to the KC which in turn, sends the transactions to the KDS for execution. Once the execution is complete, the KDS sends the results in the KDM form back to the KC. The KC routes the results to the kernel formatting system (KFS). The KFS reformats the results from the KDM form to the UDM form. The KFS then displays the results in the correct UDM form via the LIL.



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 1: The Multi-Lingual Database System (MLDS).

The four modules, LIL, KMS, KC, and KFS, are collectively known as the **language interface**. Four

interfaces with similar modules are required for the four interfacing user models and languages (i.e., relational/SQL, hierarchical/DL/I, network/CODASYL-DML, and entity-relationship/Daplex) of the MLDS.

2. The Multi-Backend Database System

The multi-backend database system (MBDS) was designed to overcome the performance problems and upgrade problems associated with the traditional approach to database system design. This goal was realized through the utilization of multiple backends connected in a parallel fashion. The backends have identical hardware, replicated software, and their own disk systems. In the multi-backend configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communications bus. Users access the system through either the hosts or the controller directly (see Figure 2).

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in database

size and transaction responses, then the MBDS produces invariant response times for the same transactions. For a more detailed discussion of MBDS the reader is referred to [Refs. 2 and 3].

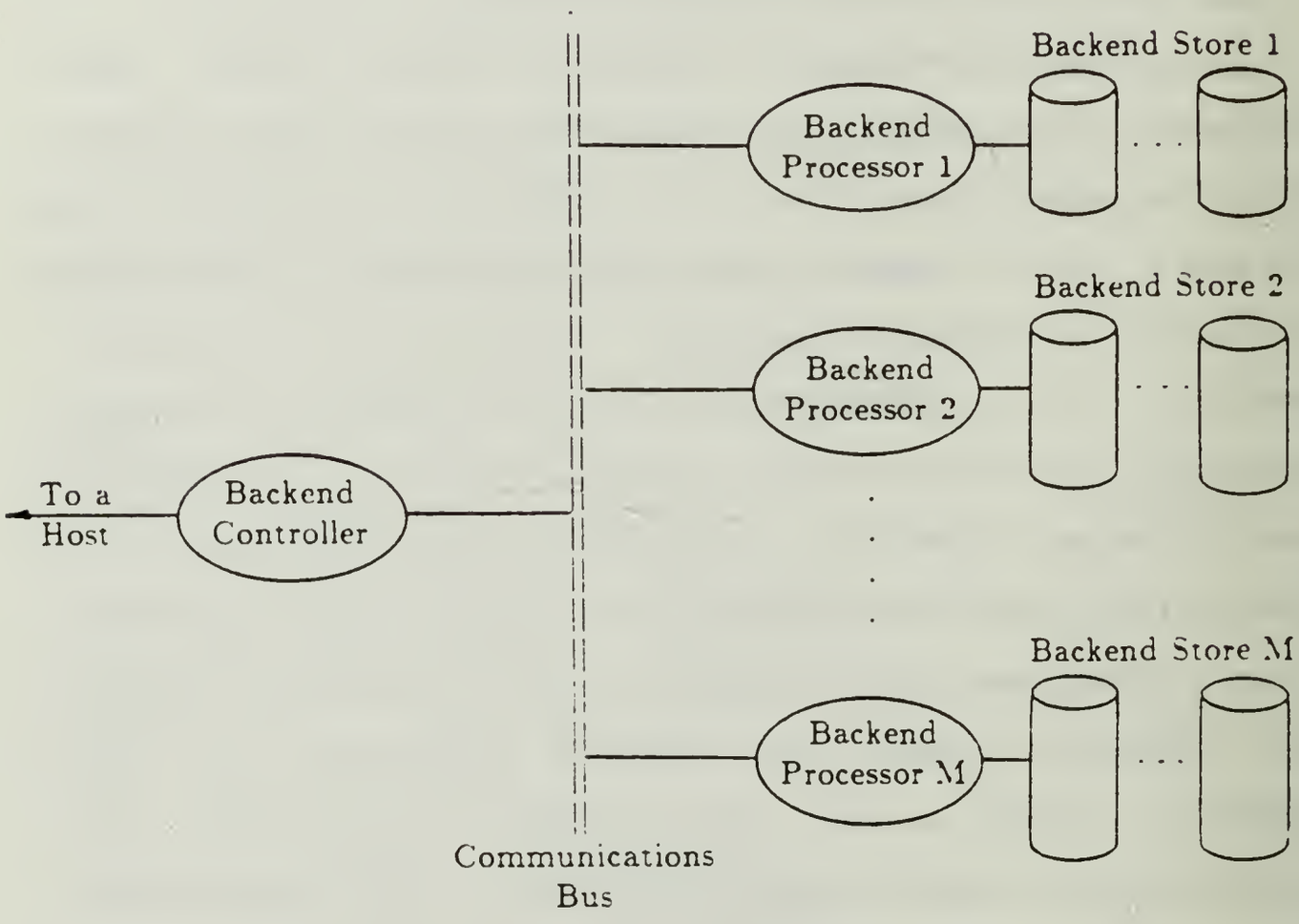


Figure 2: The Multi-backend Database System (MBDS).

In this thesis, we investigate the design of a network (CODASYL) interface for the MLDS. Banerjee [Ref. 4], provided an initial design for such an interface. We are extending his work and adapting it to support the requirements of the MLDS. In particular, we present a specification for the kernel mapping system (KMS) that will be used in the network interface. We also provide an

implementation strategy for the kernel controller (KC). The other two modules, the LIL and the KFS are nearly the same in structure as those already implemented for the DL/I and SQL interfaces, and thus, they will not be discussed in detail in this thesis. The reader is referred to [Refs. 5 and 6] for further details on the design of these modules.

Throughout this thesis, we will make extensive use of the Suppliers-and-Parts sample database used by Date [Ref. 7] for illustration of our work. The data structure diagram for this network is shown in Figure 3. There are supplier records (S), parts records (P), and shipments (SP) records. The sets of the database are suppliers-shipments (S-SP) and parts-shipments (P-SP).

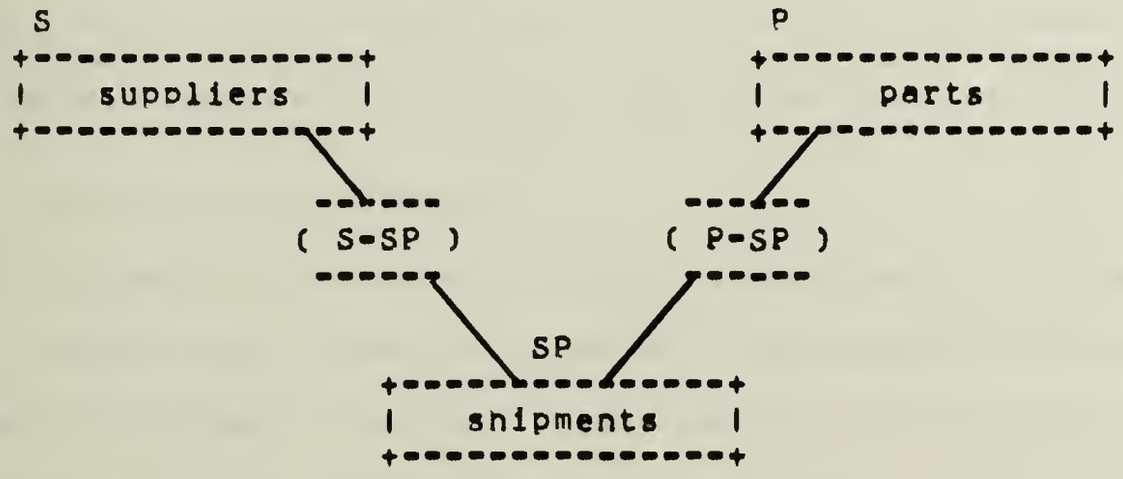


Figure 3: Data Structure Diagram of the Sample Suppliers-and-Parts Database.

In Chapter 2, we provide a description of both the network (CODASYL) data model and the attribute-based model, as well as, their associated data languages. In Chapter 3, a methodology for mapping a network (CODASYL) database into

an attribute-based database is presented. Chapter 4 is dedicated to explaining the data manipulation language translations. And, in Chapter 5, we provide implementation condiderations for both the KMS and the KC. Finally, in Chapter 6, we make our conclusions about the proposed design.

II. THE DATA MODELS

The choice of a kernel data model and a corresponding kernel data language is of vital importance in developing a multi-lingual database system. The kernel data model and the kernel data language must be capable of supporting all the necessary data model transformations and data language translations required by the MLDS language interfaces.

It is our intention in this chapter to provide a summary description of the data models related to the network (CODASYL) interface, namely the CODASYL data model and the attribute-based data model. A conceptual view of both models will be presented along with a brief discussion of the data manipulation languages associated with each model.

A. THE CODASYL DATA MODEL

In general, the network (CODASYL) data model is based on the concept of directed graphs. The nodes of the graphs usually represent entity types which are described by records, while the arcs of the graphs correspond to relationship types that are represented as connections between records. The CODASYL (Conference on Data System Languages) data model is referred to by Tsichritzis and Lochovsky [Ref. 8:pp. 119-132] as the most comprehensive specification of a network data model that exists. Thus, the

reason for choosing the CODASYL data model and its data manipulation language for the network interface of the MLDS.

1. A Conceptual View of the Model

CODASYL databases are networks of record types and set types, where records and sets are the entities which describe the databases. A record type in a CODASYL database is defined in [Ref. 4] as a collection of hierarchically related data item names or field names. These field names are specified in a schema declaration (template) for that record type. A **record** is any occurrence of a record type and has specific values assigned to the data items named in the schema declarations. This implies that a record type is simply a generic name for all of the records that are described by the same template.

Set types in a CODASYL database indicate relationships between record types. They consist of a single record type called the **owner** record type, and one or more record types called the **member** record types. Thus, a set type expresses explicit associations between different record types in the database. This characteristic makes it possible for a designer to model a large variety of real world database management problems involving diverse record types. Of special importance here is the fact that the owner record type of a set type is prohibited from being a member of the same set type.

Set types have occurrences just as record types do. Each occurrence of a set type has one occurrence of the

owner record type and zero or more occurrences of each of its member record types. The prohibition here is that a record occurrence cannot be present in two different occurrences of the same set type. This qualification emphasizes the pairwise disjointness of set occurrences of a given set type. Figure 4 gives an example of a set occurrence for the set type S-SP of our sample database.

As can be seen from the example, the CODASYL data model makes the design of a database quite simple. However, keeping track of all of the relationships can be considerably involved. Thus, one of our primary concerns in the design of a CODASYL language interface for the MLDS is to preserve these relationships without the complexity.

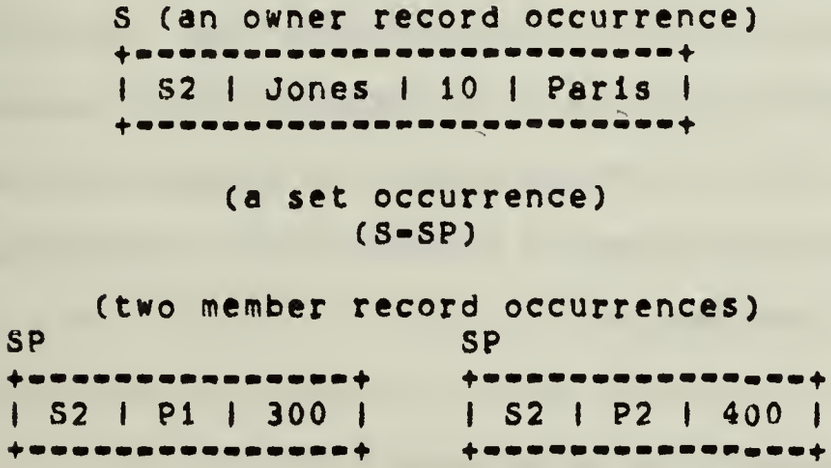


Figure 4: A CODASYL Set Occurrence.

2. The Data Manipulation Language (CODASYL-DML)

CODASYL-DML is a procedural data language. The user of a CODASYL database writes his programs in a general purpose language that hosts the CODASYL-DML. In general,

most operations in a CODASYL database are carried out by "navigating" through set occurrences. The starting point for this navigation is usually the current record of the run unit. The **run unit** is the application program (transaction) being executed. A full explanation of currency will be provided later in the thesis. Other DML operations can be based on the current record occurrence of a set type or record type.

CODASYL-DML has several primary operations which support the primary database operations of retrieval, insertion, deletion, and modification (updating existing records). Different implementations provide varying collections of these operations, but we will concentrate our discussion on the basic ones.

The cornerstone of the CODASYL-DML is the FIND statement. This statement is used to establish the currency of the run unit, and optionally used to establish the currency of the set type and the record type. The general format of the FIND statement is

FIND record-selection-expression [ ],

where the square brackets contain optional expressions for the suppression of updates to the currency indicators. In other words, we may suppress the updating of the currency for a record type, a set type, or both. The record-selection-expression has several different forms each

designed to access a particular record in three different ways, either out-of-the-blue without reference to a previously accessed record; relative to a previously accessed record; or by repetition. The other DML statements are somewhat less extravagant.

The GET statement in CODASYL-DML complements the FIND statement. Once a record is found, the GET statement places the record in the transaction's working area for access by the transaction. There are two basic formats for the GET statement. They include GET record_type, which gives the transaction access to the entire record, and GET items IN record_type, which gives access to only requested data items in the record type.

The STORE statement is used to place a new record occurrence into the database. The programmer must build up an image of the record prior to the STORE request using assignment statements which are a part of the host language in which the CODASYL-DML is embedded. Once the record image has been created, then the proper set occurrence for the record must be selected by the database management system.

The set occurrence in which the new record is stored is determined by the SET SELECTION clause specified in the schema definition for the object database. The three options available are: BY APPLICATION, which means that the application program (transaction) is responsible for selecting the correct occurrence; BY VALUE, which means the system selects the proper occurrence based on data item

values specific to the owner of the set occurrence desired; and, BY STRUCTURAL, which means that the system selects an occurrence by locating the owner record with a specific item value equal to the value of that same item in the record being stored. The restriction on the last two options is that the data items being used must have been specified with DUPLICATES NOT ALLOWED in the schema definition. A detailed discussion of syntax for the CODASYL-DML is presented later in the thesis.

If the user transaction desires to manually insert records into the database, two requirements exist. First, the schema definition must include the INSERTION IS MANUAL clause in the set desciption for this particular member record. Then the CONNECT statement is used, instead of the STORE statement, for insertion of the record into the database. The record to be inserted is the current record of the run unit. The set occurrence in which the record is inserted is determined in the same way as the STORE statement.

There is also a statement in the CODASYL-DML which performs the opposite operation, namely, the manual removal of a record occurrence from a set. The DISCONNECT statement performs this operation. It disconnects the current record of the run unit from the occurrence of the specified set that contains the record. The record occurrence still resides in the database, but it is no longer a member of the specified set. There is a qualification involved with this

statement, however. The record to be disconnected must have a RETENTION clause of OPTIONAL in the member desciption for the set type definition in the schema.

In order to delete records from a CODASYL database, the ERASE statement is used. There are four basic options to this statement; however, two of them are very complex and marginally useful, so they will not be discussed in this thesis. The simplest of the two we will deal with is the ERASE without the ALL option. This statement causes the current record of the run unit to be deleted from the database if, and only if, it is **not** the owner of a non-empty set. If it is the owner of a non-empty set, the erase fails.

The ERASE ALL option is a little less useful according to Olle [Ref. 9]. This statement causes the current record of the run unit to be deleted whether or not it is the owner of a non-empty set. Additionally, this option causes each member record of the set to be deleted, and if they too are owners of non-empty sets, their members are deleted. This action continues all the way down the hierarchy. As one can see, an entire database could be destroyed if the user is not careful when using this option.

The final statement to be covered in this thesis is the MODIFY statement. It is used to modify values of data items in a record occurrence. This includes modifying all data items or any subset of the data items in the record type. It may also be used to change the membership of a

record occurrence from one set occurrence to another, as long as, they are of the same set type. Thus, we have our basic working set of DML statements.

B. THE ATTRIBUTE-BASED DATA MODEL

The attribute-based data model was originally described by Hsiao [Ref. 10]. It is a very simple but powerful data model capable of representing many other data models without loss of information. It is this simplicity and universality that makes the attribute-based model the ideal choice as the kernel data model for the MLDS, and the attribute-based data language (ABDL) as the kernel language for the system.

1. A Conceptual View of the Model

The attribute-based data model is based on the notions of attributes, and values for these attributes. An attribute and its associated value is therefore referred to as an **attribute-value pair** or **keyword**. These attribute-value pairs are formed from a Cartesian product of the attribute names and the domains of the values for the attributes. Using this approach, any logical concept can be represented by the attribute-based model.

A **record** , in the attribute-based model represents a logical concept. In order to specify the concept thoroughly, keywords must be formed. A record then, is simply a concatenation of the resultant keywords, such that no two keywords in the record have the same attribute. Additionally, the model allows for the inclusion of textual

information, called the **record body** , in the form of a, possibly empty, string of characters describing the record or concept. The record body is not used for search purposes. Figure 5 gives the format of an attribute-based record.

```
(<attribute1,value1>, ... ,
 <attributen,valuen>,
 { text })
```

Figure 5: An Attribute-Based Record

The angled brackets, <,>, are used to enclose a keyword where the attribute is first followed by a comma and then the value of the attribute. The record body is then set apart by curly brackets, {,}. The record itself is identified by enclosure within parentheses. As can be seen from the above, this is quite a simple way of representing information.

In order to access the database, the attribute-based model employs an entity called predicates. A keyword predicate, or simply **predicate** , is a triple of the form (attribute, relational operator, value). These predicates are then combined in disjunctive normal form to produce a **query** of the database. In order to satisfy a predicate, the attribute of a keyword in a record must be identical to the attribute in the predicate. Also, the relation specified by the relational operator of the predicate must hold between the value of the predicate, and the value of the keyword. A

record satisfies a query if all predicates of the query are satisfied by certain keywords of the record. A query of two predicates

(TYPE = S) and (SNO = S4)

would be satisfied by any record of TYPE S (supplier type) whose SNO (supplier number) is S4, and it would have the form,

(<attribute1,vaue1>, ... ,<TYPE,S>, ... ,
<SNO,S4>, ... ,<attributen,valuen>,{text}).

2. The Attribute-Based Data Language (ABDL)

The ABDL as defined by Banerjee, Hsiao, and Kerr [Ref. 11] was originally developed for use with the Database Computer (DBC). This language is the kernel language used in the MLDS. The ABDL supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. Those of use to us in this portion of the MLDS work nowever, are INSERT, DELETE, UPDATE, and RETRIEVE. A user of this language issues either a request or a transaction. A **request** in the ABDL consists of a primary operation with a qualification. The **qualification** specifies the portion of the database that is to be operated on. When two or more requests are grouped together and executed sequentially, we have a **transaction** in the ABDL. There are four types of requests, corresponding to the four primary database

operations listed above. They are referred to by the same names.

Records are inserted into the database with an INSERT request. The qualification for this request is a list of keywords and a record body. Records are removed from the database by a DELETE request. The qualification for this request is a query.

When records in the database are to be modified, the UPDATE request is utilized. There are two parts to the qualification for this request. They are the query and modifier. The query specifies the records to be modified while the modifier specifies how the records are to be modified.

The final request to be mentioned here is the RETRIEVE request. As its name implies, it retrieves records from the database. The qualification for this request consists of a query, a target-list, and an optional by-clause. The query specifies the records to be retrieved. The target-list contains the output attributes whose values are required by the request, or it may contain an aggregate operation, i.e., AVG, COUNT, SUM, MIN, MAX, on one or more output attribute values. The by-clause is optional and is used to group records when an aggregate operation is specified.

As indicated, ABDL consists of some very simple database operations. These operations, nevertheless, are capable of supporting complex and comprehensive

transactions. Thus, ABDL meets the requirement of capturing all of the primary operations of a database system, and is quite useful for our purposes. Figure 6 shows examples of the four primary ABDL requests.

```
INSERT(<TYPE,SP>,<SNO,S2>,<PNO,P1>,<QTY,300>,{sample})
DELETE((TYPE = S) and (SNO = S4))
UPDATE((TYPE = SP) and (PNO = P1))(QTY = QTY + 100)
RETRIEVE((TYPE = P) and (PNAME = Nut))
```

Figure 6: Sample ABDL Requests.

III. MAPPING NETWORK (CODASYL) DATA TO ATTRIBUTE-BASED DATA

Using a modification of a procedure originally outlined by Banerjee [Ref. 4], the transformation of network data into attribute-based data becomes a relatively simple task. The data must be transformed into records which consist of a set of variable-length attribute-value pairs and a record body. The attribute-value pairs may represent the type, quantity, or characteristic of the value, and the record body is as described in the previous chapter. Additionally, all attributes in the attribute-based records are distinct, for logical reasons.

The key aspect of the mapping process is the retention of the CODASYL notions of records and sets (the linkages among records). We emphasize that the CODASYL notions of records and sets are not the same as the attribute-based notions of records and sets. Thus, the mapping algorithm presented herein uses attribute-based constructs (or notions) to implement the CODASYL notions. In the following sections, we present the various entities which must be mapped, their corresponding attribute-based equivalent, and an example of the mapping process using our sample database. It should be clear after this description, that the CODASYL notions of records and their relationships are indeed preserved in the attribute-based system.

A. THE REPRESENTATION OF A CODASYL RECORD

A CODASYL record type is structured as a hierarchical configuration of data items such as depicted in Figure 7(a), where R1 is the record name, and A, B, C, D, E, and F represent data item names. Figure 7(b) shows an occurrence of record R1. Notice that only the values of the data items are present in the CODASYL record. In the attribute-based system, both the data-item-name and its value are stored in the record.

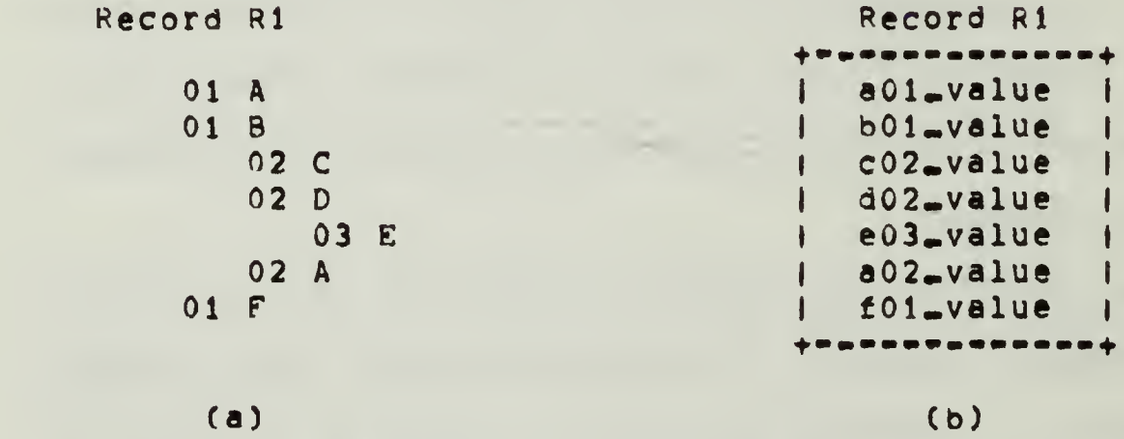


Figure 7: Hierarchical Structure of a CODASYL Record.

Thus, in order to capture the CODASYL information, keywords must be created for each of the elementary data items included in the CODASYL record. These data-item keywords should be of the form

< data_item_name,data_item_value >

where the data-item-name is qualified by data-item-names at a higher level if it is not unique. Figure 8 shows the data item representation for the CODASYL record of Figure 7.

```
(...,< A,a_value >,< B,b_value >,
     < C,c_value >,< D,d_value >,
     < E,e_value >,< B.A,b.a_value >,
     < F,f_value >,...)
```

Figure 8: Attribute-Based Representation of CODASYL Data Items.

The dots at the beginning of the record and the dots at the end of the record indicate that there are additional keywords generated for the record in order to preserve the CODASYL record information. These additional keywords are explained as follows.

Each record occurrence in a CODASYL database must also belong to a particular type. This implies that a keyword indicating record type must also be included in the attribute-based record. Its format is

```
< TYPE,record_type >
```

where TYPE is a literal.

Finally, each record occurrence of a CODASYL database has a database key (or address) generated for it. Thus, there is a requirement for representation of this value as well in the attribute-based record. The following form is used for this keyword, where DBKEY is a literal.

```
< DBKEY,database_key >
```

So, in representing record information, we have the need for three mandatory keyword types, namely, data_item_name, with or without qualification, TYPE, and DBKEY.

B. THE REPRESENTATION OF CODASYL SETS

In order for the attribute-based record to be complete, it must also include information related to CODASYL set membership, and set ordering. Since occurrences of set types are pairwise disjoint, then each member record occurrence belonging to a set occurrence is also identified by its owner record occurrence. This means that we can express set membership by inclusion of the keyword

< MEMBER,set_type,owner_database_key >

for each set occurrence in which the record is a member.

Finally, the logical position of a record occurrence within a set occurrence is often useful. Thus, ordering of member record occurrences within a set occurrence is expressed by inclusion of the keyword

< POSITION,set_type,sequence_number >

in the attribute-based record for each set in which the record is a member record.

Therefore, in representing set information, we have the need for two keyword types, those representing member records, and those representing member-record positions within sets.

C. A COMPLETE DATA-MAPPING EXAMPLE

As previously mentioned, by utilizing the above transformation scheme, we can take an existing CODASYL database and transform it into an attribute-based database without any loss of information related to the CODASYL records and sets (i.e., record relationships). The transformation should therefore result in records of the form shown in Figure 9.

```
(< TYPE,record_type >,< DBKEY,database_key >,
 < data_item_name1,data_item_value1 >,
                    .
                    .
                    .
 < data_item_namen,data_item_valuen >,
 < MEMBER,set_type1,owner_database_key1 >,
                    .
                    .
                    .
 < MEMBER,set_typep,owner_database_keyp >,
 < POSITION,set_type1,sequence_number >,
                    .
                    .
                    .
 < POSITION,set_typep,sequence_number >
 { textual information })
```

Figure 9: An Example of a Transformed CODASYL Record.

```
SCHEMA NAME IS SUPPLIERS_AND_PARTS.
 RECORD NAME IS S;
   DUPLICATES ARE NOT ALLOWED FOR SNO.
      SNO    ; TYPE IS CHARACTER 5.
      SNAME  ; TYPE IS CHARACTER 20.
      STATUS ; TYPE IS FIXED 20.
      CITY   ; TYPE IS CHARACTER 15.

 RECORD NAME IS P;
   DUPLICATES ARE NOT ALLOWED FOR PNO.
      PNO    ; TYPE IS CHARACTER 6.
      PNAME  ; TYPE IS CHARACTER 20.
      COLOR  ; TYPE IS CHARACTER 6.
      WEIGHT ; TYPE IS FIXED 4.
      CITY   ; TYPE IS CHARACTER 15.

 RECORD NAME IS SP;
   DUPLICATES ARE NOT ALLOWED FOR SNO, PNO.
      SNO    ; TYPE IS CHARACTER 5.
      PNO    ; TYPE IS CHARACTER 6.
      QTY    ; TYPE IS FIXED 5.

 SET NAME IS S_SP;
   OWNER IS S;
      ORDER IS SORTED BY DEFINED KEYS
      DUPLICATES ARE NOT ALLOWED.
   MEMBER IS SP;
      INSERTION IS AUTOMATIC
      RETENTION IS FIXED;
      KEY IS ASCENDING PNO IN SP;
      SET SELECTION IS BY VALUE OF SNO IN S.

 SET NAME IS P_SP;
   OWNER IS P;
      ORDER IS SORTED BY DEFINED KEYS
      DUPLICATES ARE NOT ALLOWED.
   MEMBER IS SP;
      INSERTION IS AUTOMATIC
      RETENTION IS FIXED;
      KEY IS ASCENDING SNO IN SP;
      SET SELECTION IS BY VALUE OF PNO IN P.
```

Figure 10: Schema for the Suppliers-and-Parts Database.

In order to demonstrate the transformation process further, Figure 10 above provides the schema definition for our sample Suppliers-and-Parts database. Using this schema

definition, the CODASYL record occurrences of Figure 11 are transformed into the attribute-based records of Figure 12.

```
S
+--------------------------+
| S2 | Jones | 10 | Paris |
+--------------------------+

P
+-------------------------------+
| P1 | Nut | Red | 12 | London |
+-------------------------------+

SP
+---------------+
| S2 | P1 | 300 |
+---------------+
```

Figure 11: Sample Record Occurrences from the Suppliers-and-Parts Database.

```
(<TYPE,S>,<DBKEY,1>,
 <SNO,S2>,<SNAME,Jones>,
 <STATUS,10>,<CITY,Paris>,
 { Sample supplier record })

(<TYPE,P>,<DBKEY,2>,
 <PNO,P1>,<PNAME,Nut>,
 <COLOR,Red>,<WEIGHT,12>,
 <CITY,London>,
 { Sample parts record })

(<TYPE,SP>,<DBKEY,3>,
 <SNO,S2>,<PNO,P1>,
 <QTY,300>,
 <MEMBER,S_SP,1>,
 <MEMBER,P_SP,2>,
 <POSITION,S_SP,1>,
 <POSITION,P_SP,1>,
 { Sample SP record where the record
    belongs to two different sets })
```

Figure 12: Attribute-Based Equivalent of Record Occurrences in Figure 11.

IV. MAPPING CODASYL-DML STATEMENTS TO ABDL REQUESTS

Having demonstrated how network databases can be successfully transformed into attribute-based databases, we are now ready to examine the mapping of network data manipulation statements into ABDL requests. As mentioned in Chapter 2, the CODASYL data manipulation language will be used for the MLDS network interface. It should be noted here though, that only a subset of all the available DML statements will be used in the MLDS network interface. Specifically, the following CODASYL statements will be incorporated in this stage of the project: FIND, GET, STORE, CONNECT, DISCONNECT, ERASE, and MODIFY. Of these, only the useful formats were considered for the MLDS. It should be further noted that the syntax for these various statements was derived from the syntax presented by Date, Olle, and the original CODASYL report [Refs. 7, 9, and 12], respectively.

In this section we discuss each of the above statements and their associated mapping process. Prior to describing the mapping, however, we first explain the notion of currency in a CODASYL database, and introduce the data structures that are necessary to carry out the mapping process. The Appendix, the KMS (Kernel Mapping System) specification, gives a detailed look at the mapping process

and the specific algorithms applied to accomplish the language translations.

A. THE NOTION OF CURRENCY

In general, the above data manipulation statements can be grouped into two categories, data retrieval statements and data updating statements. However, the common thread between the two groups, as well as, the individual functionality of each statement, depends quite heavily on the notion of **currency** among the records and sets of the CODASYL database.

The concept of currency in a CODASYL database can be compared to the well known concept of current position in a file. The idea here is that for each application program being run on the system, a table of "currency indicators" is maintained. In general, the currency indicator is an object whose value is a **database key**. It serves as a "cursor" which points to either a record or a set under consideration by the application program. Database keys are values generated by the database management system that uniquely identify each individual record in the database.

The currency indicator table for a given application program (or run unit) identifies the record occurrence "most recently accessed" by the run unit for each of the following: each type of record, each type of set, "any type" of record, and each type of realm (Realm is a CODASYL concept that will not be considered in this thesis.) "Any

type" of record refers to the most recently accessed record occurrence, no matter what its type is. This record is appropriately called, the **current of the run unit** , and is the most important currency of all. Additionally, the **current of the set type** may be either an owner record or a member record, whichever was accessed most recently.

B. DATA STRUCTURES NECESSARY FOR ACCURATE TRANSLATION

1. The Currency Indicator Table (CIT)

A currency indicator table (CIT) is created for each application program that is run using the MLDS network interface. These tables are dynamic in nature. They are instantiated upon the first call to the database system, and are updated as subsequent CODASYL-DML calls are made to the database system.

```
CIT
  RUN_UNIT
    record_type
    database_key

  record_type(i)
    database_key

  set_type(i)
    boolean (is record an owner record)
    record_type
    database_key
    member_record_type
    owner_record_type
    owner_database_key
```

Figure 13: Information Contained in the CIT.

The CIT contains an entry for the current of run unit, the current of record_type for each record_type in the

database, and the current of set_type for each set_type in the database. Each entry in the CIT should contain at least the information shown in Figure 13 as suggested by Meyer [Ref. 13].

2. The Request Buffer (RB)

When mapping the CODASYL-DML statements to ABDL requests, there are one-to-many correspondences between the two types of statements. Thus, for each CODASYL-DML statement, several ABDL requests may have to be generated to assemble the necessary information for accurate execution of the translated CODASYL-DML statement. In other words, a series of ABDL requests may be generated for each CODASYL-DML statement. Some of the requests are initially incomplete, however, and require information returned by previous RETRIEVE requests which are a part of that statement's translation. This implies the need for storage of intermediate information for the requests.

The request buffer (RB) acts as that storage mechanism for information returned by what we term, auxiliary retrieve requests (ARR'S). There must be one RB for each RETRIEVE request issued. The exact role that each buffer plays is explained in the next section of this chapter. In general though, upon successful execution of an ARR, all record occurrences satisfying the request are maintained in the buffer. This information is then used for subsequent request execution.

C. MAPPING THE FIND STATEMENTS TO THE ABDL RETRIEVES

The general format of the CODASYL FIND statement is

FIND record_selection_expression [ ],

while the general format of the ABDL RETRIEVE is

RETRIEVE Query Target-list [by Attributes].

As previously stated, there are several formats for the FIND statement, each with a different functionality. Some of these, however, are thought to be considerably more useful than others, so we only concern ourselves with the ones of most value in the MLDS. Before proceeding, the reader should note that in CODASYL statements, upper-case notation represents literals, lower-case represents user supplied variable names, and square brackets indicate optional clauses. We now examine the mapping process for each of the CODASYL statements to be included in the MLDS network interface.

1. The FIND ANY Statement

The FIND ANY statement tells the database system to locate any record of type, record_type1, whose values for item1 through itemn match those in that record's template in the user work area. The syntax for the FIND ANY is:

```
FIND ANY record_type1 USING item1, ... ,itemn
IN record_type1.
```

To perform the mapping of this statement, the kernel mapping

system (KMS) must first substitute the word RETRIEVE for the words FIND ANY. Then the KMS must form a predicate, (TYPE = record_type1), for inclusion in the final query. The next step in the process requires the KMS to determine the values that the search is to be based on. These values are found in record_type1's record template.

After acquiring these values, the KMS then forms additional predicates for the data items specified in the original statement, and includes these predicates in the query. Since all of the necessary information is available to the KMS for this particular CODASYL statement, there is no need for an auxilliary retrieve request (ARR). However, an RB is needed to store the retrieved data once the request has been executed.

With the query now formed, the KMS creates a target-list to complete the RETRIEVE request. The target-list consists of all attributes of the requested record. Thus, the translated CODASYL-DML statement is:

```
RETRIEVE (( TYPE = record_type1) and
          (item1 = user_value1) and
                .
                :                and

          (itemn = user_valuen))
          ( all attributes ) [ by DBKEY ],
```

This request is then passed to the KC of the interface for execution. An example utilizing our sample database will help to illustrate the mechanics of the mapping process.

The requirement is to find any Supplier record, S, where that supplier's city is 'Cleveland'. The CODASYL procedure is:

```
MOVE 'Cleveland' TO CITY IN S
FIND ANY S USING CITY IN S
```

(Note: The MOVE statement is an assignment statement found in the host COBOL language.) The KMS would respond to this series of code by performing the following actions:

Step 1: 'Cleveland' is placed in the S template for the attribute CITY.

Step 2: A RETRIEVE request is formed as such:

```
RETRIEVE ((TYPE = S) and
          (CITY = 'Cleveland'))
          (SNO, SNAME, STATUS, CITY)
          by DBKEY
```

Step 3: The KMS passes the request to the KC for execution.

This operation results in having all S records satisfying the query ((TYPE = S) and (CITY = 'Cleveland')) placed in the request buffer and sorted according to the value of the database keys. Figure 14 shows the contents of Buf1 after the RETRIEVE is executed.

```
+-----------------------------+
|                             |
| <S6,Mathews,25,Cleveland>   |
| <S8,Jones,30,Cleveland>     |
|                             |
+-----------------------------+
```

Figure 14: Contents of Buf1 After RETRIEVE.

Upon issuance of a GET statement by the user, the first record in the RB is returned, provided the RETRIEVE has been successful.

2. The FIND CURRENT Statement

The FIND CURRENT statement is a rather simple one in that no direct mapping to an ABDL request is necessary. This statement is used to change the current of run unit indicator from its present value to the value of the database key of the current record of set_type1. Thus, the interface has the responsibility of updating the current of run unit indicator (i.e., CIT.RUN_UNIT.type <-- record_type1 and CIT.RUN_UNIT.dbkey <-- dbkey of current of set_type1). The syntax for this statement is:

FIND CURRENT record_type1 WITHIN set_type1

As an example of this process, suppose we desire to start a search at the current SP occurrence in set_type S-SP. The CODASYL statement would be:

FIND CURRENT SP WITHIN S-SP

After encountering this statement, the KMS passes the update information on to the KC for execution. The KC then updates the currency indicators to reflect the changes. The current of run unit becomes the current SP record occurrence of the current S-SP set occurrence.

3. The FIND DUPLICATE WITHIN Statement

The FIND DUPLICATE statement is used for sequential access within a particular set occurrence. It locates the first record_type1 record within the current set_type1 occurrence whose values for item1 through itemn match those of the current record of set_type1. The syntax used for this statement is:

```
FIND DUPLICATE WITHIN set_type1 USING
item1, ... ,itemn IN record_type1
```

The mapping process for this request assumes that the records being requested are already in an RB. Therefore, no RETRIEVE request is generated for this statement. Instead, the KMS forwards the set type, record type, and the data item name(s), on which the search is based, to the KC. The KC then takes this information, and locates the RB containing the set. It then compares the specified data item values for the current record of the set type to each of the other member records until the first duplicate record within the set is found. This record is made available for return to the user. The CIT is then updated to reflect the new currency status. This approach is advantageous, in that, all of the records for a particular set occurrence are already available in an RB, eliminating the need for further accesses to the database in the event of subsequent requests for duplicate records, such as would be the case in a loop.

The following example illustrates the mapping process: Find the next shipment record for supplier S1 in which the quantity shipped is 100. A possible CODASYL procedure for accomplishing this consists of the following statements:

```
MOVE 'S1' TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 100 TO QTY IN SP
FIND SP WITHIN S-SP CURRENT USING QTY IN SP
FIND DUPLICATE WITHIN S-SP USING QTY IN SP
```

The effect of the first four statements is to locate the first SP occurrence for supplier S1 that has a QTY of 100. The next statement finds the next SP record in the S-SP set with the same QTY, namely, 100.

The interface would respond to the FIND DUPLICATE request as follows:

Step 1: Execution of the first four statements produces the results in the RB of Figure 15.

Step 2: The KC then gets the value of the data item, QTY, by going to the RB and finding the current record of the S-SP set using the record_type and set_type information given.

Step 3: The KC now locates the next record in the set with QTY = 100 and makes it ready for return to the user.

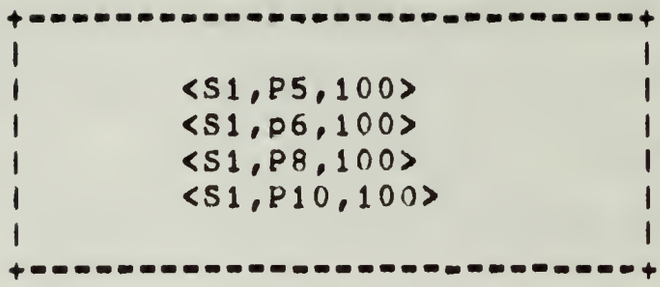


Figure 15: Contents of Buf1.

4. The FIND FIRST Statement

The FIND FIRST statement locates the first member record of a set occurrence. This statement has several other forms: FIND LAST, FIND NEXT, and FIND PRIOR. Since they are all mapped in exactly the same way, we only describe the mapping process for the FIND FIRST. The syntax for the FIND FIRST is:

FIND FIRST record_type1 WITHIN set_type1

Upon encountering the FIND FIRST, the KMS must ensure that record_type1 is a member record type of set_type1. This is necessary, since this particular FIND is based on the currency indicators, and the current of set_type1 may be an owner record, as noted earlier when discussing currency of set types. Assuming that the current record of set_type1 is a member record, the KMS then forms a RETRIEVE request that will retrieve every member record of the current set_type1 occurrence into its RB. The interface would then only have to return the first record in the set in order to satisfy the request. If the statement had been FIND LAST, the last record in the set would be returned.

The response would be similar for the FIND NEXT and FIND PRIOR statements. Assuming that the set occurrence has already been retrieved into an RB, the interface would simply locate the current record of set_type1 in the RB and return the record after it in the case of FIND NEXT, or the record before, it in the case of the FIND PRIOR. The fact that all of the member records of the set occurrence are already in an RB, eliminates the need for additional database accesses. Thus, the only ABDL request that need be formed is this:

```
RETRIEVE ((TYPE = record_type1) and
          (MEMBER,set_type1 = owner_dbkey_set-type1))
          (all attributes) [by DBKEY]
```

As an example, consider the following request: Find all the part numbers (PNO's) for parts supplied by supplier S4. A possible CODASYL procedure to accomplish this would be:

```
MOVE 'S4' TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 'NO' TO EOF
FIND FIRST SP WITHIN S-SP
PERFORM UNTIL EOF = 'YES'
  GET SP
  (add PNO IN SP to result list)
  FIND NEXT SP WITHIN S-SP
END_PERFORM
```

The statements of concern here are the FIND FIRST and the FIND NEXT. The reader need only be aware that in CODASYL only one record at a time is made available to the user. Thus, the need for the perform loop.

In response to the above sequence of statements, the interface would perform these steps:

Step 1: The KMS of the interface would form a RETRIEVE request to get all members of the S-SP set owned by supplier S4. Since each record has a predicate which identifies them as members of a particular set occurrence, the task is fairly easy. The request is:

```
RETRIEVE ((TYPE = SP) and
          (MEMBER,S-SP = dbkey of S4))
          (SNO,PNO,QTY) [by PNO].
```

The results of executing this request are shown in Figure 16. We can see that every member record of the set has been fetched from the database and is available for return to the user. The FIND FIRST causes the first record to be returned to the user.

Step 2: Since the CODASYL procedure has a FIND NEXT statement, the same RB is used. In other words, the KC does not need to execute a new retrieve request. It merely makes available the next record in the RB until all records have been returned to the user as per the loop.

Since we are only looking for PNO values, the interim user code would specify the attribute to be returned and the interface would respond accordingly.

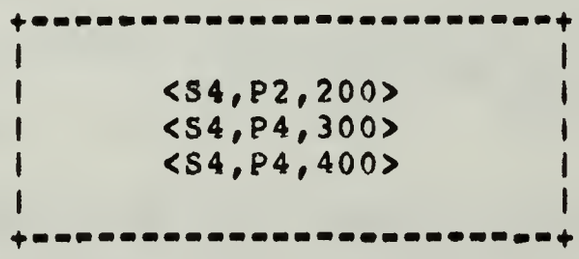


Figure 16: Contents of Buf1 After RETRIEVE.

5. The FIND OWNER Statement

The FIND OWNER statement causes the owner of the current of set_type occurrence to be returned to the user. The syntax for this statement is:

FIND OWNER WITHIN set_type1

The mapping of this statement is relatively straightforward. The KMS must simply form a RETRIEVE request based on information available in the CIT. The KMS examines the CIT entry for set_type1 and extracts the owner's type and database key value directly from the table. It is then an easy task to form the request:

```
RETRIEVE ((TYPE = owner of set_type1) and
          (DBKEY = owner dbkey of set_type1))
          (all attributes)
```

As an example, suppose we want to know the STATUS of the supplier for part number P6. Let us assume that previous statements have set up the current S-SP set occurrence to be S2/P6/20. The CODASYL statement is:

FIND OWNER WITHIN S-SP.

In response to this request, the interface takes the following action:

Step 1: The KMS forms the request:

```
RETRIEVE ((TYPE = S) and
          (DBKEY = dbkey of S2))
          (SNO,SNAME,STATUS,CITY)
```

Step 2: The KC would cause the execution of the above request, resulting in an RB containing one record, namely, the S2 record.

Based on the interim user code, the STATUS value is returned to the user from the RB by the interface.

6. The FIND WITHIN CURRENT Statement

This statement causes the first record within the current occurrence of set_type1 whose values for item1 through itemn match those in the user work area for record_type1. The following syntax is used for this statement.

```
FIND record_type1 WITHIN set_type1 CURRENT
USING item1, ... ,itemn IN record_type1
```

This statement is similar to the FIND DUPLICATE except that the search values are obtained from the user vice the current record of set type. Thus, only a single RETRIEVE request is needed. That request takes the form:

```
RETRIEVE ((TPYE = record_type1) and
          (MEMBER.set_type1 = dbkey of owner set_type1)
          and (item1 = user value1)
          and ...
          and (itemn = user valuen))
          (all attributes) [by DBKEY]
```

This request is then passed to the KC for execution. If there is more than one record satisfying this query, the RB for the request contains them all. However, only the first record encountered is returned to the user.

To illustrate the process of this mapping, we return to a previous example: FIND the first shipment for supplier

S1 in which the quantity is 100. Using the first four statements from the example in section C.3 we have,

```
MOVE S1 TO SNO IN S
FIND ANY S USING SNO IN S
MOVE 100 TO QTY IN SP
FIND SP WITHIN S-SP
  CURRENT USING QTY IN SP.
```

In order to carry out this request, the following steps are taken by the interface:

Step 1: The KMS forms the request:

```
RETRIEVE ((TYPE = SP)
          and (MEMBER,S-SP = dbkey of S1)
          and (QTY = 100))
          (SNO,PNO,QTY) [by DBKEY]
```

Step 2: The KC executes the above request and causes the first record in the RB to be made available to the user.

D. MAPPING THE CODASYL GET STATEMENTS

The GET statements in the CODASYL-DML can be considered as data retrieval statements just as the FIND statements are, except that the GET request can only access records that have been previously identified by a FIND statement. It is the statement that actually gives the user access to the individual records. There are three options available with the GET statement, and we examine each in turn. In developing these mappings, we decided not to directly map the GET statements to ABDL RETRIEVE's, but to simply issue instructions to the KC for handling them.

1. **The GET and GET record_type Statements**

The GET statement, without the specification of a particular record type, causes the entire current record of run unit, that is, every data field in the record, to be returned to the user via the User Work Area (UWA). In tne MLDS network interface, recogniton of this statment by the KMS results in the following response:

Step 1: The KMS informs the KC that the "next" available record in the RB that contains records of the type CIT.RUN_UNIT.type is to be passed to the user. Note that the type of the current of run unit does not matter in this case.

The GET record_type statement is identical to the GET option alone, with the exception that the user specifies a particular record type. In this case, the KMS must determine if the types of the current of run unit matches the record type specified before issuing instructions to the KC. Also, every data item is returned to the user.

Returning to our example in section C.4, the "GET SP" statement causes the return of the record, <S4,P2,200>, to the user the first time the GET is issued and each of the other records in sequence as the loop continues.

2. **The GET item1, ... ,itemn Statement**

Unlike the other GET options, this statement causes specific data items to be returned to the user. The syntax of the statement is:

GET item1, ... ,itemn IN record_type1.

The KMS must compare the record_type to the current of run unit and also ensure that the data items listed match the data items in the record type specified. Once this is done and is successful, the KMS issues instructions to the KC just as in the above case. Only this time, specific data items are returned from the records accessed.

As an example, suppose we wanted only the PNO values from the SP records. The value returned from our last example would be P2, with subsequent GET statements returning each PNO value in succession.

E. MAPPING THE DATA-UPDATING STATEMENTS

In this section, we examine the CONNECT, DISCONNECT, STORE, MODIFY, and ERASE statements. At this point, the reader should have a basic understanding of the mapping process as previously described. Thus, for the sake of brevity, the reader is referred to [Refs. 7, 9, and 12] for detailed descriptions of the statements and any restrictions involved with their use. We therefore, confine our discussion of these statements to a broad definition, the mapping process itself, and in most cases, an example.

1. The CONNECT Statement

The CONNECT statement is used for manual insertion of the current record of run unit into the current occurrences of the set type(s) specified. The syntax is:

CONNECT record_type1 TO set_type1, ... ,set_typen.

This statement requires that the record_type1 record be a member of the sets specified and also have an insertion clause of MANUAL for those sets.

The CONNECT statement maps directly to the ABDL UPDATE request. The UPDATE format is:

UPDATE Query Modifier

In the case of the CONNECT, the mapping is very simple. First, the KMS replaces CONNECT by the word UPDATE. Then, the type and database key of the record to be inserted is taken from the CIT to form the query ((TYPE = record_type1) and (DBKEY = CIT,RUN_UNIT,dbkey)). Finally, in order to construct the modifier, the KMS get the database key of the owner of the current occurrence set_type1 from the CIT. The KMS then forms the modifier, (MEMBER,set_type1 = CIT,set_type1,owner_dbkey) for each set type specified. Each set type specified has its own complete UPDATE request generated.

One might ask, why use an UPDATE instead of an INSERT request. Well, the difference is that the CONNECT statement involves records already in the database. And, because the keyword, <MEMBER,set_type,NULL>, is in the record whose connection value is NULL, it becomes a simple matter to just update that particular keyword, thereby connecting the record. We recall that in an attribute-based database, keywords, not pointers, are used to connect one record to another. The INSERT statement on the other hand,

involves records not already in the database. Thus, the completely translated CONNECT statement is:

```
UPDATE ((TYPE = record_type1) and
       (DBKEY = CIT.RUN_UNIT.dbkey))
       (MEMBER.set_type1 = CIT.set_type1.owner_dbkey)
```

2. The DISCONNECT Statement

The DISCONNECT is just the opposite of the CONNECT statement. It causes the current record of the run unit to be disconnected from the set listed. The set occurrences selected are determined by the current of set type indicators. Since several set types may be listed in the statement, only one statement is needed in order to make several removals. The records still remain in the database. They are simply disconnected from specific sets. The syntax is:

```
DISCONNECT record_type1 FROM
set_type1, ... ,set_typen.
```

The DISCONNECT statement requires that record_type1 be a member of the set types listed, and that the record be removed from the set occurrences that are current. Because of the way we represent set membership in the attribute-based record, this task is very simple. Since we are disconnecting the current of run unit, and it contains the database keys of the owners of the set occurrences it belongs to, and since each record can only be in one occurrence of the same set type (pairwise disjointness), the

mapping process is direct. We simply form an UPDATE request for each set type listed. Thus, the keyword, <MEMBER,set_type1,owner_dbkey>, is modified, and becomes the keyword, <MEMBER,set_type1,NULL>. To accomplish this, the KMS forms the request,

```
UPDATE ((TYPE = record_type1) and
        (DBKEY = CIT,RUN_UNIT,dbkey))
        (MEMBER,set_type1 = NULL)
```

and passes it to the KC for execution.

3. The MODIFY Statement

The MODIFY statement causes the entire current record of the run unit to be modified or specific data items in that record to be modified. The syntax is either,

```
MODIFY record_type1, or
MODIFY item1, ... ,itemn IN record_type1.
```

This statement also, has a rather straightforward mapping to the ABDL UPDATE request. The statement assumes that the user has supplied the necessary data item values for modification in record_type1's record template in the UWA. Therefore, the job of the KMS portion of the interface is to get this user supplied information and form the following UPDATE request for each data item to be modified:

```
UPDATE ((TYPE = record_type1) and
        (DBKEY = CIT,RUN_UNIT,dbkey))
        (data itemi = user value for i).
```

As an example of this process, consider changing the STATUS

and CITY attributes of supplier S4 from 20 and 'London' to 15 and 'Chicago', respectively. The CODASYL request is:

```
MOVE S4 TO SNO IN S
MOVE 15 TO STATUS IN S
MOVE 'Chicago' TO CITY IN S
FIND ANY S USING SNO IN S
MODIFY STATUS,CITY IN S.
```

(Note: The SNO numbers in this example are unique.) Once again the MOVE statements set up the S record template for use by holding the new values for the S4 record. The FIND statement establishes the S4 record as the current record of the run unit. The KMS then responds to the MODIFY statement by forming the following two UPDATE requests and passing them to the KC for execution.

```
UPDATE ((TYPE = S) and
        (DBKEY = dbkey of S4))
        (STATUS = 15)

UPDATE ((TYPE = s) and
        (DBKEY = dbkey of S4))
        (CITY = 'Chicago')
```

If the entire record was to be changed, the first option would have been used, requiring the KMS to form an UPDATE request for each data item in the S record type.

4. **The STORE Statements**

The STORE statement is used in the CODASYL-DML to insert a new record into the database. Before a new record can be inserted though, it must be constructed. This takes place in the UWA. The syntax for the STORE is:

```
STORE record_type1
```

In mapping the STORE statement, care must be exercised in determining the proper set occurrence in which to place a record, if it is a member record. This is necessary only in the case of automatic insertion. The interface must have access to the original database description in order to determine the set selection criterion for each new record to be inserted. The three criterion are: by APPLICATION, by STRUCTURAL, and by VALUE. Each of these requires a slightly different mapping. Therefore, we examine each individually.

In addition to the set selection criterion, the interface must determine if any data items of the record being inserted has a DUPLICATES NOT ALLOWED clause assigned to it. In the case that such data items exist, the interface must form a RETRIEVE request to determine the existence of records in the database that may already have items with the same value as those in the record that is about to be stored. Thus, each STORE statement consists of at least one ABDL RETRIEVE and one ABDL INSERT request. We shall see, however, that additional RETRIEVE's are necessary for the set selection criterion of STRUCTURAL and VALUE.

a. The STORE-by-Application Statement

This method of set selection assumes that the proper occurrences of sets are indicated in the CIT. Therefore, the KMS forms two requests, the RETRIEVE to

determine the status of duplicates, and the INSERT request which stores the record. The process is as follows:

Step 1: The KMS forms the RETRIEVE below with the search based on all data items designated to have DUPLICATES NOT ALLOWED. The values for these items in the new record are supplied by the user via the UWA record template.

```
RETRIEVE((TYPE = record_type1) and
         (data item1 = user value1))
         (DBKEY) [by DBKEY]
```

Step 2: The KMS forms the INSERT request

```
INSERT(<TYPE,record_type1>,<DBKEY,***>,
       <data item1,user value1>,
       <MEMBER,set-type1,set_type1,owner_dbkey>)
```

and forwards both requests to the KC for execution.

Step 3: The KC issues the RETRIEVE request. If the RB returns with no DBKEYs, then the INSERT request is executed. Otherwise, the INSERT is not executed and an error condition exists.

b. The STORE-by-Value Statement

The by-VALUE set selection criterion means that the set occurrence we need has a data item whose value is equal to the value in the specified UWA record template which has that data item as one of its fields. The reader is referred to Figure 10 for the syntax of the set selection clause of sets S-SP and P-SP, as examples. This type of STORE requires that the data item in question have a DUPLICATES NOT ALLOWED clause also, and that the user initialize the data item in its UWA record template before issuing the STORE request.

The by-VALUE criterion therefore, places the additional requirement on the interface of locating the

owner of the proper set occurrence before the new record can be inserted into the database. This is accomplished by the issuance of a second RETRIEVE request by the KC if the first RETRIEVE, as mentioned above, returns NULL. The steps in the process are:

Step 1: The KMS forms the first RETRIEVE as above. Then for each set type in which the new record is a member, a RETRIEVE request is formed to get the owner database key. The request is:

```
RETRIEVE((TYPE = owner type) and
        (search item = user value))
        (DBKEY) [by DBKEY],
```

Step 2: The KMS forms the following INSERT request:

```
INSERT(<TYPE,record_typei>,
       <DBKEY,***>,
       <data itemi,user valuei>,
       <MEMBER,set_typei,***>)
```

Step 3: The KC executes the first RETRIEVE to determine if duplicates exist. If not, the remaining RETRIEVEs are executed in turn to get the database keys of the owners of the set occurrences to which the new record belongs. Once these values are returned, the KC finishes building the INSERT request, and executes it.

c. The STORE-by-Structure Statement

The by-STRUCTURAL set selection criterion is similar to by-VALUE except that the proper set occurrence is selected by comparing a data item value in one record type to the value of that same data item in another record type. From our sample database, we could have a by-STRUCTURAL clause indicating that the SNO value in S must equal the SNO value in SP. Thus, we must search for an SP record

occurrence with the same SNO value as that in the S template in the UWA. Once again, this data item must have a DUPLICATES NOT ALLOWED specification.

The mapping here is identical to that for the by-VALUE case except that the second through ith RETRIEVEs are based on equality of values in seperate records. Since the idea is the same, we will not give the specifics of the mapping here. It is presented in detail in the Appendix.

5. The ERASE Statements

The ERASE is the final CODASYL-DML statement we consider for the MLDS network interface. As implied, it is the statement that causes deletion of records from the database. There are two options with this statement, as previously discussed in Chapter 2. We begin with the simple ERASE. The syntax for this ERASE statement is:

ERASE record_type1

The ERASE without the ALL option deletes one record from the database, namely, the current record of the run unit. The only requirement is that the record is not the owner of a non-empty set. This means that in mapping this statement, we need to issue a RETRIEVE request prior to the deletion request to determine if there are any sets whose members are connected to this record. Therefore, for each set type in which the current of run unit is an owner, we have a predicate in the RETRIEVE query of the form: (MEMBER.set_type1 = CIT.RUN_UNIT.dbkey). The request is:

```
RETRIEVE((MEMBER.set_type1 = CIT.RUN_UNIT.dbkey)and
                           :
                           .                        and

        (MEMBER.set_typen = CIT.RUN_UNIT.dbkey))
        (DBKEY) [by DBKEY].
```

The next step in the mapping process is to form a DELETE request that deletes the current of run unit. That request is:

```
DELETE((TYPE = CIT.RUN_UNIT.type) and
       (DBKEY = CIT.RUN_UNIT.dbkey)).
```

So, the KMS in this case issues two ABDL requests to the KC for execution. The KC would execute the RETRIEVE first. If it results in a NULL RB, then the DELETE is executed. Otherwise, the ERASE fails.

The second ERASE under consideration is the ERASE with the ALL option. The ERASE ALL syntax is:

```
ERASE ALL record_type1.
```

As mentioned in Chapter 2, this option is like a "vacuum cleaner" in that it deletes every record in the hierarchy starting with the current record of the run unit. The difference between the mapping of this statement and the previous ERASE is that, RETRIEVEs must be formed to get the database keys of each member of every set that the current of run unit owns, and then RETRIEVEs are formed recursively thereafter for the members of lower level sets until the

leaves of the hierarchy are reached. In addition to these RETRIEVE requests, a DELETE request is needed for each member of every set connected to the current of run unit and the current of run unit itself. As one can see, this could become quite complex. Therefore, we briefly decribe the algorithm, and refer the reader to the Appendix for the details.

In mapping the ERASE ALL, the KMS forms a REIRIEVE request to get each member of every set owned by the current of run unit. It then forms a DELETE for each of these members. Once it has taken care of the first level, the KMS proceeds to form requests which erase all of the descendents in the same fashion by calling a recursive procedure called "erase_all". Finally, the KMS forms a DELETE request to delete the current record of the run unit as in the previous ERASE. This concludes the desciptions of the mapping process from CODASYL-DML to ABDL.

V. IMPLEMENTATION CONSIDERATIONS

In Chapter 1, we provide a brief description of the four modules included in the CODASYL language interface, namely, the language interface layer (LIL), the kernel mapping system (KMS), the kernel controller (KC), and the kernel formatting system (KFS). In this chapter, we present considerations for the implementation of the KMS and the KC.

A. THE KERNEL MAPPING SYSTEM (KMS)

The KMS is the second module in the MLDS CODASYL interface. It is called from the language interface layer (LIL) when the LIL receives CODASYL input requests from the user. In this section, we discuss the specification of the KMS (see Appendix) for the network (CODASYL) interface. We describe its operation, present a conceptual view of its data structures, and give an example of the KMS translation process. Implementations of the KMS for the DL/I and SQL interfaces can be found in [Ref. 5:pp. 45-80] and [Ref. 6:pp. 47-68], respectively. These implementations provided the basic framework for the design of the CODASYL KMS.

The KMS must perform the following functions: (1) parse the request to validate the user's CODASYL syntax, and (2) translate, or map, the request to equivalent ABDL requests. Once the necessary ABDL requests have been formed, they are made available to the kernel controller (KC) for execution.

1. The KMS Parser/Translator

The grammar-driven parser is the most important aspect of the KMS. The Yet-Another-Compiler Compiler (YACC) [Ref. 14] is an ideal choice for the construction of the parser. YACC is a program generator designed for syntactic processing of token streams. YACC functions as follows: It must be given a specification of the input language structure (a set of grammar rules), the code tnat is to be invoked when the grammar rules are recognized, and a low-level input routine that generates tokens from a regular expression input. Given these inputs, YACC generates a program that syntactically recognizes the input language, and causes specific user code to be invoked, as required, throughout the recogniton process. The user's code here is the code that performs the CODASYL-DML to ABDL translation. The Lexical Analyzer Generator (LEX) [Ref. 15] is the low-level input routine that we propose. LEX is a program generator designed for lexical processing of input character streams. It takes regular expressions as input, and generates a program that partitions the input stream into tokens. These tokens are then output to the parser for further processing.

The parser produced by YACC consists of a finite-state automaton with a stack. It performs a top-down parse, with left-to-right scan and one token look-ahead. Control flow within the parser begins at tne highest-level grammar rule. It then descends through the grammar, hierarchically,

calling lower- and lower-level grammar rules which search for the appropriate tokens in the input streams. As these tokens are recognized, some portions of the mapping/translation code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a higher-level grammar rule is satisfied. Once a rule is satisfied, further translation can be accomplished. When all of the necessary low-level grammar rules have been satisfied, and control has propagated back up to the highest-level rule, the parsing and mapping process is complete. In section B, we provide an example of the parsing and translation process.

2. The KMS Data Structures

The KMS needs several different data structures. However, we confine our discussion here to the structures which carry the information necessary for the proper execution of the translated requests. The structures that fall into this category, are the CIT structure, and the request nodes which are passed to the KC for execution. A description of the minimum requirements for these structures is given below.

The CIT is described in Chapter 4. This structure carries all of the currency information for a particular run unit, and is vital to the proper translation and execution of CODASYL statements. The LIL of the interface initializes the CIT. The KMS has read access to the CIT at all times, while all updates of the CIT are done by the KC only. In

the following sections, we discuss each of the data structures that are directly related to the parsing and translation process.

a. The 'find_node' Data Structure

The find_node is created and used any time that a CODASYL FIND statement is mapped by the KMS. Since we are considering the implementation of six different FIND formats, we must ensure that the find_node has at least four fields, one identifying the node as a find_node, a second, specifying the type of FIND statement that must be executed, i.e., FIND ANY, FIND CURRENT, FIND OWNER, and FIND WITHIN, one field to indicate the set type involved, and one field to identify the record type used in the statement.

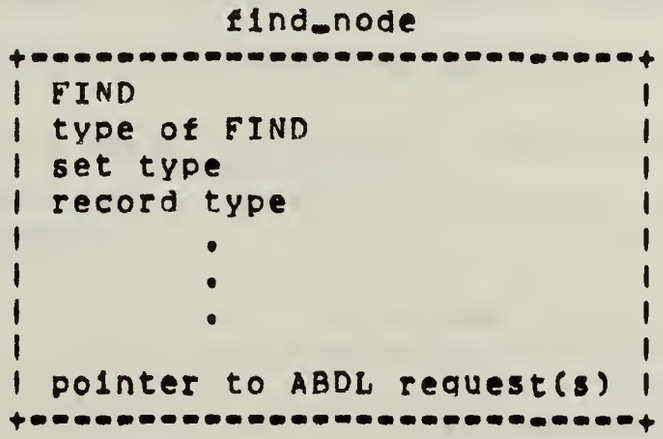


Figure 17: The 'find_node' Data Structure.

In addition to the above information, each find_node must also have a field which contains a pointer to the specific ABDL request that resulted from the mapping process. With regard to the FIND CURRENT request and the FIND DUPLICATE request, no ABDL request is generated.

Therefore, the pointer would be NULL. Figure 17 above illustrates the type of structure described, where the dots represent any additional implementation-dependent information which might need to be included.

b. The 'get_node' Data Structure

The get node carries the information that the KC needs in order to return the proper data to the user. It must have a field identifying it as a get node and a field identifying the type of GET format being used. Additionally, a field identifying the record type in question must also be included. In the case of the GET item_list format, the node should include a pointer to a list of data item values to be returned. If the format is GET record_type, the pointer field would be NULL, and the KC would return all attributes of the record. The same is true for the simple GET format. Figure 18 is an example of this type of structure.

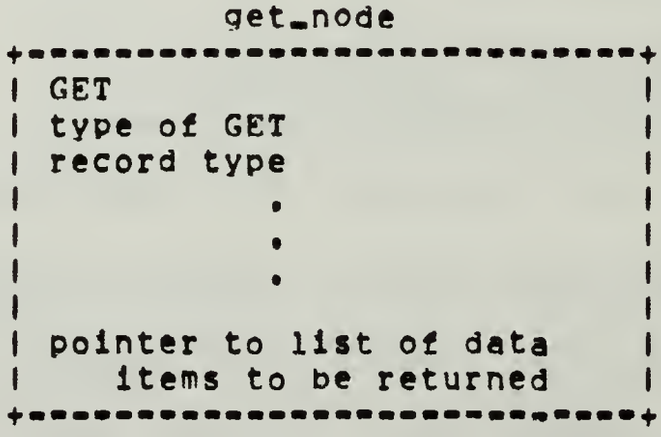


Figure 18: The 'get_node' Data Structure.

c. The 'connect_node' Data Structure

The connect_node is created and used whenever a CONNECT statement is mapped by the KMS. There are two primary fields in this node. The first field identifies the node as a connect_node. The second field is a pointer to the list of ABDL UPDATE requests generated by the KMS during its grammar-driven parse. This list may contain one or more requests depending upon the number of sets that the record must be connected to, as described in Chapter 4. Under the current implementation of the MBDS, a seperate UPDATE request must be executed for each attribute in a record that is to be changed. Thus, the need for multiple UPDATE requests. Recall, that the attribute to be changed in this case is the MEMBER,set_type attribute. Figure 19 shows the basic structure for this node.

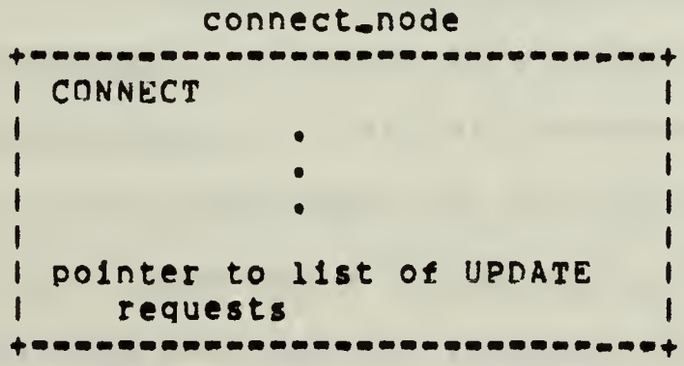


Figure 19: The 'connect_node' Data Structure.

d. The 'disconnect_node' Data Structure

The disconnect_node is created and used whenever a DISCONNECT statement is encountered by the KMS. The fields of this node are exactly the same as those of the

connect_node. In this case however, the value of the attribute MEMBER.set_type is set to NULL, disconnecting the record from designated set occurrences. Once again, we have an identifier field, and a list of UPDATE requests.

e. The 'modify_node' Data Structure

As with the disconnect_node, the modify_node is also very similar to the connect_node. It is created whenever a MODIFY statement is encountered by the KMS. It has two fields, an identifier field, and a pointer to a list of UPDATE requests. The UPDATE requests in the modify_node are used to alter the value of specific data item attributes within a particular record. The number of requests on this list can vary from one to the maximum number of data items in the record, depending on the MODIFY format chosen by the user.

f. The 'store_node' Data Structure

The store_node is the most interesting of the data structures presented so far. It must contain at least four fields. The first is the identifier field. The second field is a pointer to a RETRIEVE request. This request is generated by the KMS in order to determine the existence of duplicate values for data items declared to have DUPLICATES NOT ALLOWED in the database schema. The third field of importance relates to the set selection criterion for the record being stored. It is generated to retrieve the owner database key(s) of the proper set occurrence(s) for the new

record. This request is only generated in the cases of the by-VALUE and by-STRUCTURAL set selection criterions.

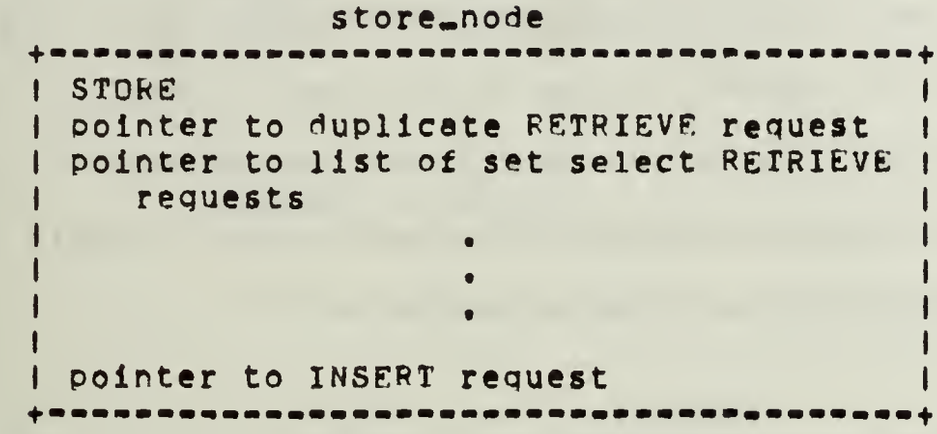


Figure 20: The 'store_node' Data Structure.

The final field required for the store_node is a pointer to the INSERT request which will actually cause the record to be placed into the database. Figure 20 above is an illustration of this data structure. As mentioned before, the dots in the figure represent additional implementation-dependent information. Should the set selection criterion be by APPLICATION, the second RETRIEVE pointer would be NULL.

g. The 'erase_node' Data Structure

The final data structure we discuss is the erase_node. This node is created whenever an ERASE statement is mapped by the KMS. If the ERASE without the ALL option is mapped, the erase_node must contain the following four fields. First, it must contain an identifier field. Second, it must contain a type field with a value of NULL, indicating that it does not have the ALL option. The

third field in this node must be a pointer to the RETRIEVE request that will determine if the record being deleted owns a non-empty set. Should this request return NULL, the KC would execute the request stored in the fourth field. This is the field containing a pointer to the DELETE request that will delete the current record of the run unit. Figure 21 gives a representation of this structure.

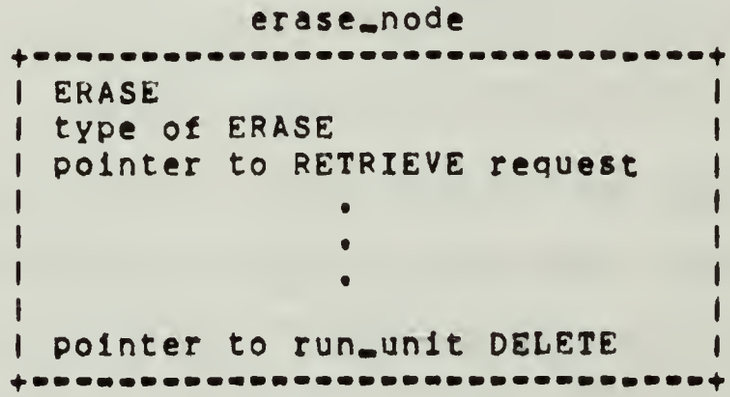


Figure 21: The 'erase_node' Without the ALL Option.

The erase_node created for the ERASE with the ALL option will be considerably more complex than the previous case. First, there must be an identifier field and a type field. Then there must be two pointer fields. The first pointer field will point to the list of RETRIEVE requests generated to get all the descendents of the record being deleted. The second pointer field will point to the list of DELETE requests generated for each of the descendent records. Finally, the last field in the structure should be a pointer to the DELETE request that deletes the current record of the run unit. Figure 22 is an example of this structure.

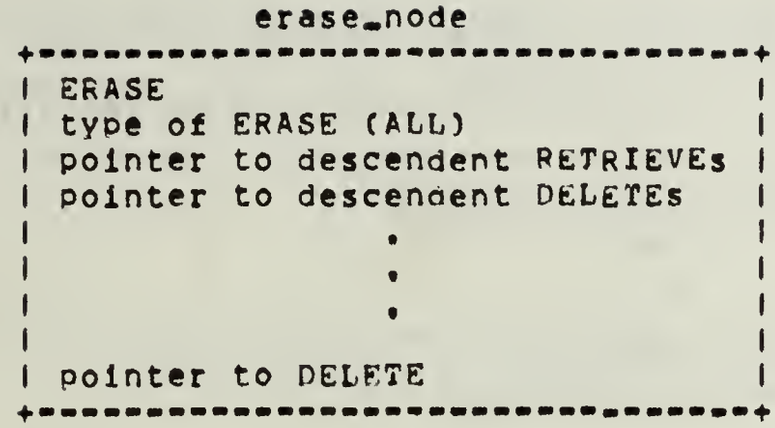


Figure 22: The 'erase_node' With ALL Option.

B. THE MAPPING PROCESS: AN EXAMPLE

In this section, we present an illustrative example of the parsing and translation processes within the KMS. Recall from the previous chapter, that not all of the features of CODASYL are incorporated in our specification. Additionally, since we have thoroughly covered the mappings in the previous chapter, we do not discuss these translations in detail.

As an example of the KMS mapping process, we use a simple CODASYL MODIFY request. We begin our example by showing the dml_statement portion of the KMS. We then step through the grammar and demonstrate relevant portions of our design in a system specification language (SSL). The reader should note that throughout the example, we only show the portions of the SSL that would actually be executed. The entire KMS design is shown in the Appendix. The portion of the grammar relevant to this example is shown in Figure 23.

In Figure 23, we have included the grammar rules only, and not the code to be invoked as each rule is satisfied.

```
statement: ddl_statement
         | dml
         ;

dml: dml_statement
   | dml dml_statement
   ;

dml_statement: set_flag
             | move
             | get
             | find
             | store
             | connect
             | disconnect
             | erase
             | modify
             | perform_loop
             | if_then
             ;

modify: MODIFY item_list IN record_type
      | MODIFY record_type
      ;

item_list: item_name
         | item_list COMMA item_name
         ;

record_type: IDENTIFIER
           ;

item_name: IDENTIFIER
         ;
```

Figure 23: The KMS dml_statement Grammar.

The source CODASYL procedure we use for our example is:

```
MOVE 100 TO QTY IN SP
MODIFY QTY IN SP.
```

Before giving the ABDL equivalent of this request, however, we must make the assumption that the record being modified

is the current record of the run unit. We also assume that the database key for this record is 10. With these two assumptions in mind, the ABDL equivalent of the MODIFY statement is:

```
[UPDATE ((TEMPLATE = SP) and (DBKEY = 10))
        (QTY = 100)],
```

For the sake of brevity in our example, we will not go through the mapping process for the MOVE statement. The reader need only be aware that the new value for the attribute QTY in the record template for record type SP, has been set to the value, 100, by the previous parse/translation. Now, we may proceed with the mapping of the MODIFY statement.

At the beginning of the mapping process, the parse descends the grammar hierarchy searching for tokens in the grammar rules which match those in the input. Notice that the first rule to be tried is the ddl_statement rule. As the parse descends the ddl_statement rule, hierarchically, there are no tokens which match the example input stream. Thus, the ddl_statement rule is not satisfied, and the parse begins again at the dml rule.

When the dml rule is called, it immediately calls the dml_statement rule,

```
dml_statement: set_flag
             | move
             | get
             | find
             | store
             | connect
             | disconnect
             | erase
             | modify
             | perform_loop
             | if_then
              ; .
```

The dml_statement rule then calls the set_flag rule. The set_flag rule is not satisfied, however, and the move rule is called. It too, is not satisfied. So, the process of checking each successive rule is continued until we reach the following rule:

```
modify: MODIFY
          {
          select_list = NULL
          }
        item_list IN record_type
          {
             /* error checks are made here */
           alloc and init new 'modify' node
                ...

           for (each data_item on select_list)
              alloc and init new abdl_str
              /* form UPDATE request */
            copy "[UPDATE ((TEMPLATE = 'record_type')
              and (DBKEY = CIT.RUN_UNIT.dbkey))"
              to abdl_str
            get 'item_value' from move_list
            concat "('data_item' = 'item_value')]
              to abdl_str
            connect abdl_str to 'modify' node
           end_for
          end_else
          }
      | MODIFY record_type
       ; .
```

The modify rule looks for the token, MODIFY, in the input. Since it is present, the first portion of the rule matches,

and the code following the token in the rule is invoked. This code simply resets a local list which will eventually contain the names of the data items which are to be modified.

The next rule called is the item_list rule. This rule searches for a list of identifiers in the input by calling the item_name rule, and recursively calling item_list, as indicated. In our example, the single identifier, QTY, satisfies the first portion of this rule, namely, item_name. Thus, the item_list rule is satisfied. The syntax for the item_list rule is:

```
item_list: item_name
             {
             add the item_name
               to select_list
             }
           item_list COMMA item_name
             {
             add succesive
               item_name to selec_list
             }
         ;  .
```

The next portion of the modify rule is the token, IN. This token matches the token, IN, in the input stream, and the parse continues. Finally, the last portion of the modify rule which must be satisfied is the record_type rule. This rule is satisfied by matching the identifier, SP, in the input, with the token, IDENTIFIER, in the record_type rule. After matching these two, the entire modify rule is

satisfied, and the invocation of the remaining mapping and translation code following the rule takes place.

The mapping and translation occurs as follows. First, a series of checks are made to determine if the record being modified is the current record of the run unit, if the new value(s) have been placed in the record template, and if all of the items identified in the item list belong to the record in question. Then, a modify node is created. Following this step, an UPDATE request is generated for each of the data items being modified. Finally, each of the update requests are connected to the modify node for execution by the KC. With the mapping and translation code executed, the modify rule is completely satisfied, and control propagates back up the grammar hierarchy to the dml rule which checks for more input.

As one can see, quite a significant amount of work is done by the KMS in preparing requests for use by the KC. We feel that by adequately providing information to the KC, we greatly reduce the amount of work that must be done by the KC. This means less coding for the implementor and should lead to less complexity in the KC.

C. THE KERNEL CONTROLLER (KC)

The KC is the third module in the MLDS CODASYL interface. It is called by the language interface layer (LIL) when a new database is being loaded, and is called by the kernel mapping system (KMS) when an existing database is

being manipulated. The KC is the module which performs the task of controlling the submission of ABDL transactions to the multi-backend database system (MBDS) for processing. Implementations of the KC for the DL/I and SQL interfaces can be found in [Ref. 5:pp. 84-105] and [Ref. 6:pp. 69-88], respectively.

The KC must perform the following functions: (1) submit transactions to the MBDS, (2) receive and store results of transactions, (3) update the currency indicator table, and (4) cause the proper data to be returned to the user.

1. The Structure of the KC

Because of the large number of types of transactions that the KC must process, we suggest that the overall structure of the KC be based on the "case" control structure. At the top of the control structure is a master control procedure which is responsible for initialization of variables, pointers, and data structures, as well as, deciding the type of ABDL transaction that is being processed. Recall that there are two major types of transactions, creation of a new database, and manipulation of an existing database. Thus, a two element case is required in the master control procedure. These cases are then used to call subsequent procedures and functions which handle the transactions which fall under the above categories.

a. Creation of a New Database

The creation of a new database is the least difficult transaction that the KC will handle. It involves loading the CODASYL schema created by the KMS into the KDS (MBDS), in its attribute-based form. It is also responsible for mass storage of new records during a database creation transaction. Thus, the KC must also assign database keys to the new records throughout this process.

Currently, work is being done on the algorithms necessary to accomplish the transformations above and the mass storage requirement. This work will not be completed in time for inclusion in this thesis. Suffice it to say, however, that once the work is completed, the only requirement of the KC in this case, is to call a procedure to load the database schema, call a procedure to load the database descriptor file, and then call a procedure to load the new database. Once these procedures are executed, control is returned to the LIL.

b. Manipulation of an Existing Database

The manipulation of an existing database can also be divided into sub-cases. There are the data retrieval requests and the database update requests. However, all of these can be handled by a single case structure. Recall that each time the KC is called, a request node of some type is made available to the KC. These request nodes are then used to determine which option within the case structure to execute. The structure is

illustrated in Figure 24. In the following sections, we present the procedural requirements for each type of data manipulation transaction.

```
case op_type of

  create_db: call load_schema
             call load_descriptors
             call load_db_recs
             ;
  find: case type of
          any: call find_any
          current: call find_current
          duplicate: call find_duplicate
          (first,last,
           next,prior): call find_conseq
          owner: call find_owner
        end_case;
  (connect,
   disconnect,modify): call update_db
                                  ;
  store: call store_rec
         ;
  erase: call delete_recs
         ;
  get: call get_rec
       ;
```

Figure 24: The KC Control Structure.

(1) **The FIND Procedures.** There are six basic types of FIND requests utilized in our system. The first of these is the FIND ANY request. Upon encountering a find_node whose type field is ANY, the find_any procedure is called. This procedure sets up the request buffer to receive any results that may be returned. It then issues the request to the KDS. Upon return from the KDS, the find_any procedure must update the CIT, based on the type and database key of the record that is the first record in

the request buffer. A pointer is then set to point to this record in preparation for returning it to the user. The record is not returned though, unless the user issues a GET request.

If the request is a FIND CURRENT request, the find_current procedure is called. Its job is quite easy. It must simply update the CIT, by setting the current of run unit indicator to the type and database key of the current record of the set type specified in the request node.

When the request is a FIND DUPLICATE request, the find_duplicate procedure is called. This procedure assumes that the records being requested are already in a request buffer. Thus, the only information required from the find_node is the record_type being searched for, the set_type of interest, and the data item values on which the search is based. The procedure locates the request buffer, and sets a pointer to point to the first duplicate record found. This record then becomes the record returned when the user issues a GET request. The procedure also updates the CIT accordingly.

The next type of FIND request is the FIND FIRST, LAST, NEXT, or PRIOR. In these cases, if the type is first, last, next, or prior, the find_conseq procedure is called. It bases its performance on the type of the find_node. If the type is next or prior, the procedure assumes that the records are already in a request buffer.

It looks for the correct request buffer based on the record_type specified in the find_node, and sets a pointer to point to the next or prior record relative to the current record of the set type this buffer is holding. In other words, each record in the buffer is a member of the current set type occurrence, and the find_conseq procedure simply points to the record before or after the current record for that set. Once again, this is the record returned when the user issues a GET request.

If the type of the FIND is first or last, the find_conseq procedure does the following. First, it checks to see if a request buffer exists for the set type requested in the find_node. If no such buffer exists, the procedure creates a request buffer and issues the RETRIEVE request attached to the find_node. The results of the request are then placed in the new request buffer, and a pointer is set to point to the "first" or "last" record in the set for return to the user.

The next type of FIND request is the FIND OWNER. When this is the type of the find_node, the find_owner procedure is called. It's function is fairly straightforward. A request buffer is created to hold the record that is the owner record of the set type indicated in the find_node. The find_owner procedure then issues the RETRIEVE request attached to the find_node, and prepares the record for return to the user.

The final type of FIND request, expected by the KC, is the FIND WITHIN CURRENT request. In this case, the find_within procedure is called. The procedure creates a request buffer for the storage of records returned and issues the RETRIEVE request associated with the current find_node. Again, a pointer is set to point to the first record in the request buffer in order that this record might be returned when a GET request is issued by the user. It should be noted that in each case above, the CIT is updated unless a currency suppression list has been attached to the find_node in question.

(2) **The CONNECT, DISCONNECT, and MODIFY Procedures.** The CONNECT, DISCONNECT, and MODIFY requests are handled by the KC in the same general manner. When either a modify_node, a connect_node, or a disconnect_node is encountered by the KC, the procedure, update_db, is called. If the node is a modify_node, the KC simply submits the attached ABDL UPDATE requests to the KDS for execution. After execution, control is returned to the LIL.

If the node passed to proceddure update_db is a connect_node or a disconnect_node, all of the above applies, except that before giving control to the LIL, the KC must update the CIT. When a record is connected to a set type, that record becomes the current record of the set type. When a record is disconnected from a set type, the entry in for that set type in the CIT is set to NULL and remains so until another record of the set type is accessed.

(3) **The STORE Procedure.** When the KC recognizes a store_node, the procedure store_rec is called. The first task performed by store_rec is to execute the first RETRIEVE request attached to the node. This request determines if there are records in the database which have attribute values that are not to be duplicated. If the request buffer created for this RETRIEVE is **non-empty** at the end of execution, there is an error. If the request buffer is **empty** , then store_rec performs in the following manner. For each RETRIEVE request on the set select RETRIEVE list, a file buffer is created and the RETRIEVE request is issued. These requests return the database keys of the owners of the set occurrences to which the new record belongs.

After execution of the set select RETRIEVE list, the procedure store_rec then assigns a database key to the new record, and proceeds to complete the INSERT request attached to the store_node. It is very important that the order in which the database keys are accessed from the request buffer match the order of the attributes, MEMBER.set_typei, in the INSERT request. The INSERT request is then issued. Now, because we have not accessed this record previously, and it has become the current of run unit, store_rec must provide a buffer to hold this record in case a GET request is issued immediately following the STORE request. An example of this process is warranted at this point. Suppose, we desire to store the SP occurrence, S5/P6/700. The CODASYL sequence might be:

```
MOVE 'S5' TO SNO IN SP
MOVE 'P6' TO PNO IN SP
MOVE 700 TO QTY IN SP
MOVE 'S5' TO SNO IN S
MOVE 'P6' TO PNO IN P
STORE SP
```

The first three MOVEs initialize the new record's data values. The next two MOVE's are used to aid in determining which S and P occurrences the new record belongs to, because its set insertion mode was declared to be automatic. The KMS takes this information and the STORE SP statement, and produces a store_node containing the following:

```
(Duplicates RETRIEVE request)
RETRIEVE ((TYPE = SP) and (SNO = S5) and (PNO = P6))
         (DBKEY)

(List of RETRIEVEs to get owner DBKEYs)
RETRIEVE ((TYPE = S) and (SNO = S5)) (DBKEY)
  RETRIEVE ((TYPE = P) and (PNO = P6)) (DBKEY)

(INSERT request for new record)
INSERT (<TYPE,SP>,<DBKEY,***>,<SNO,S5>,<PNO,P6>,
        <MEMBER,S-SP,****>,<MEMBER,P-SP,****>)
```

We assume, for the sake of our example, that the DBKEYs for the owner records are 10(S) and 12(P), and that the DBKEY of the new record is 98. We also assume that there is no duplicate SP record in the database. So, when store_rec issues the first RETRIEVE, the request buffer returned is empty.

Store_rec then proceeds to execute the list of RETRIEVEs that return the owner DBKEYs of the new record. It creates a request buffer for the first RETRIEVE on the

list and issues the request. Once the first request is executed, a buffer is created for the second request, and that request is executed. The issuance of these RETRIEVEs produces the results in the request buffers depicted in Figure 25. The procedure store_rec now takes the new DBKEY value, and the information from the request buffers and completes the INSERT request.

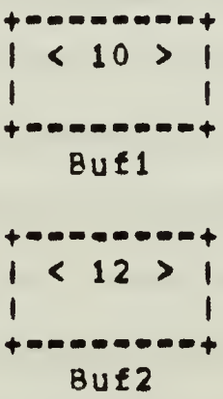


Figure 25: Buf1 and Buf2 After Execution.

The final form of the INSERT request is:

```
INSERT (<TYPE,SP>,<DBKEY,98>,<SNO,S5>,<PNO,P6>,
        <MEMBER,S-SP,10>,<MEMBER,P-SP,12>)
```

The INSERT request is then issued to the KDS. If no currency suppression list is attached to the store_node, the CIT is updated to reflect a change in the S-SP and P-SP currency as well as, the current of run unit. A request buffer is also created, and the record is stored in the buffer. As one can see, the store_rec procedure can be a very comprehensive one.

(4) The ERASE Procedures. The ERASE request is handled by the procedure, delete_recs. If the type of the erase_node is NULL, then delet_recs proceeds in the following manner. First, a request buffer is created, and the RETRIEVE request attached to the erase_node is issued to the KDS. This request determines if the record being deleted is an owner of a non-empty set. If the request buffer is not empty after execution of the RETRIEVE, then the erase fails, and we have an error condition. If the request buffer is empty after execution of the RETRIEVE, then delete_recs issues the DELETE request attached to the erase_node. This request deletes the current record of the run unit. After the deletions, delete_recs updates the CIT by setting the current of run unit indicator to NULL.

Should the type of the erase_node be ALL, we have a different sequence of events. The delete-recs procedure must create request buffers for each RETRIEVE request on the descendent retrieve list in the erase_node. It must then issue each of these RETRIEVEs storing the results, returned DBKEYs, in the proper request buffer. After the list of RETRIEVEs has been issued, the delete_recs procedure then completes the DELETE requests attached to the erase_node and issues each DELETE request to the KDS for execution. Once again, the CIT is updated to reflect the change in currency i.e., current of set_types become NULL as appropriate.

(5) **The GET Procedure.** The GET request is handled by the get_rec procedure. This procedure has a relatively easy task. It simply looks at the type of the get_node, examines the record_type involved, and retrieves either the entire record of specific fields of the record from the request buffer in which the record resides. The GET request operates on the current of run unit. So, the request buffer in question should be the request buffer containing the current record of the run unit, provided the current record of the run unit is not NULL. Finally, the reader should note that with each of the above procedures, deallocation of request buffers when they are no longer required, is also an important consideration in this process.

VI. CONCLUSIONS

As mentioned in the introduction, the conventional approach to database system development has resulted in numerous single-model, single-language systems with little, if any, flexibility or extensibility. In addition, these systems are slow compared to the system proposed by this research effort. Our system, the multi-lingual database system (MLDS), provides an alternative to the development of seperate stand-alone database systems which use single data models. The MLDS will bring flexibility, extensibility, and efficiency to the world of database management. The MLDS will be able to execute transactions written in any of four well-known and important data languages, namely, DL/I, SQL, CODASYL, and Daplex.

In this thesis, we have presented a methodology for supporting network database management within the MLDS. Specifically, we have provided a data model transformation strategy, and a data language translation strategy for the network data model and the CODASYL data language, respectively. We have presented a design specification for the kernel mapping system (KMS) to be used in the CODASYL interface. A discussion of the concepts involved and the data structures necessary for the interface to work properly has also been presented.

One of the design goals of this project was to make the interface as compatible as possible with the designs of the DL/I and SQL interfaces in order to fully utilize existing software. The Daplex interface is not mentioned here, because it is being developed in parallel with the CODASYL interface. By pursuing this goal, we also eliminate the need for changes in the MBDS and the ABDL. Thus, it is recommended that the implementation of the CODASYL interface follow closely, the implementations of the DL/I and SQL interfaces. The implementor(s) should pay particular attention to any commonalities between funtions and data structures.

We feel that the work presented herein is sufficient for implementation of the CODASYL interface. All that remains is for the code to be written, and placed in the host computer. Once the CODASYL interface and the Daplex interface have been completely implemented, the system should be tested as a complete system for projected efficiency, effectiveness, and responsiveness to user needs. It is anticipated that this research and development effort will ultimately result in a new era for database management that will allow for increased productivity and profitability in the marketplace.

Currency Indicator Table

References made in the following specification to CIT refer to the Currency Indicator Table. This table consists of structures that hold information identifying the current record of record-type, set-type, and run-unit (run-unit is the application program being run). The following is the proposed structure for this table [Ref. 13].

```
struct  CIT
  {
   struct  RUN-UNIT       *run;
   struct  rec-type-node   *next-rec-type;
   struct  set-type-node   *next-set-type;
  }

struct  RUN-UNIT
  {
   char  rec-type[   ];
   int   dbkey;
  }
```

For each record type in schema:

```
struct  rec-type-node
  {
   char                 type[   ];
   int                 dbkey;
   struct rec-type-node   *next-rec-type;
  }
```

For each set type in schema:

```
struct  set-type-node
  {
   boolean              OWNER;
   char                 TYPE[   ];
   int                 dbkey;
   char                 member[   ];
   char                 owner[   ];
   int                 owner-dbkey;
   struct set-type-node   *next-set-type;
```

```
%{
   boolean: first-move = TRUE /* flag for MOVE operation */
   boolean: first-time    /* general purpose flag */
   boolean: sys-flag-value /* boolean value of system flags */
   ptr: curr-temp-rec     /* ptr to last record added to move-list */
   ptr: curr-temp-item    /* ptr to next item node to be added to
                          record-template node of movelist */
   list: suppression-list  /* list of record types and/or set types */
                          for which currency updates are suppressed */
   list: select-list      /* list of data items used for record section */
   list: connect-list      /* list of sets to which current of run
                          unit is to be connected or disconnected */
   list: tgt-list         /* list of attribute names to be accessed */
   list: move-list         /* list of record templates used with
                          MOVE statement */
   list: curr-non-dup-list /* list of data items for which duplicates
                          are not allowed in current record-type */
   int: level-number      /* level of data item in record types */
   char: member-type      /* string variable to hold a name */
%}
```

```
start  statement

statement: ddl-statement
       | dml
         ;

dml: dml-statement
  | dml dml-statement
    ;

ddl-statement: schema-defn  record-list  set-list
            ;

schema-defn: SCHEMA NAME IS schema-name SEMI-COLON
            {
            locate db-id schema header node
            if (db names do not match)
              print ("Error-given db-name doesn't
                    match name in file")
              perform yyerror()
              return
            end-if
            initialize db-key /* starting value is 1 */
            }
          ;

record-list: record-desc
            {
            set db-id node ndn-first-rec ptr
            }
        | record-list  record-desc
            {
            connect successive record nodes
            }
          ;

record-desc: record  data-item-list
            {
            curr-non-dup-list = NULL
            }
          ;

record: RECORD  NAME  IS
        {
        allocate and init a new
          record node (NREC-NODE)
        allocate curr-non-dup-list
        db-id-node ndn-num-rec++
        }
      record-spec
      ;
```

```
record-spec: record-type
           {
           if (record-type not defined yet)
             copy record-type to current
               record node (NREC-NODE)
             make this the current record node
           end-if
           else
             print ("Error-'record-type' record
                   doubly defined")
             perform yyerror()
             return
           end-else
           }
         SEMI-COLON  duplicates-list
         ;


set-list: empty
       | set-desc
          {
          set db-id node  ndn-first-set ptr
          }
       | set-list set-desc
          {
          connect successive set node(s)
          }
        ;


set-desc: set-desig  owner-spec  member-spec
        ;

set-desig: SET NAME IS
           {
           allocate and init a new set node (NSET-NODE)
           db-id-node ndn-num-set++
           }
         set-type
           {
           if (set-type not yet defined)
             copy set-type to current set node (nsn-name)
             establish curr-set-ptr
           end-if
           else
             print ("Error-'set-type' set doubly defined in db")
             perform yyerror()
           end-else
           }
         SEMI-COLON
         ;
```

```
owner-spec: OWNER IS  aa  SEMI-COLON
          ;


aa: record-type
     {
     if (record-type not defined)
       print ("Error-'record-type' record does not exist")
       perform yyerror()
       return
     end-if
     else
       copy record-type to current set node (nsn-owner-name)
       locate record-type node
       nsn-owner(ptr) = record-type node
     end-else
     }
  | SYSTEM
    ;


member-spec: MEMBER IS  record-type
            {
            if (record-type not defined)
              print ("Error-'record-type' record does not exist")
              perform yyerror()
              return
            end-if
            else
              copy record-type to current set node (nsn-member-name)
              locate record-type node
              nsn-member(ptr) = record-type node
            end-else
            }
           SEMI-COLON insert-clause retention-clause
            {
            alloc set-select node
            }
           set-select-clause SEMI-COLON
           ;

duplicates-list: empty
           | dupl SEMI-COLON
             ;

dupl: duplicate-spec
    | dupl  duplicate-spec
      ;


duplicate-spec: DUPLICATES ARE NOT ALLOWED FOR item-spec
             ;
```

```
item-spec: item-name
          {
          alloc new non-dup node
          copy item-name to non-dup node
          add non-dup node to curr-non-dup-list
          }
        | item-spec COMMA item-name
          {
          alloc successive non-dup nodes
          copy successive item-names to non-dup nodes
          add successive non-dup nodes to curr-non-dup-list
          }
         ;


data-item-list: item-desc
              {
              connect new attr-node to record-node
              }
            | data-item-list item-desc
              {
              connect successive attr-node(s) to record-node
              }
             ;


item-desc: level-num
          {
          allocate and init a new attr-node (NATTR-NODE)
          NATTR-NODE nan-level-num = level-number
          record-node nrn-num-attr++
          }
        data-item-desc
          {
          if (nan-level-num = level number of current attribute node)
            connect new attr node to current attr node
            if (nan-level-number > 1)
              connect nan-parent ptr of new node
            end-if
          end-if
          else if (nan-level-number > level number of current attr node)
            connect nan-child ptr of current attr node to new attr node
            connect nan-parent ptr of new attr node to current attr node
          end-else-if
          else
            locate last attr node with same level number
            set that node's nan-next-attr ptr to the new attr node
            update current attr pointer
          end-else
          }
        ;
```

```
data-item-desc: item-name
                {
                copy item-name to attr-node (NATTR-NODE)
                if (item-name not on curr-non-dup-list)
                  attr-node nan-dup-flag = 1
                end-if
                }
              SEMI-COLON  data-type  PERIOD
              ;


level-num: empty
           {
           level-number = 1   /* default value */
           }
         | INTEGER
           {
           level-number = INTEGER
           }
          ;


data-type: CHARACTER INTEGER
           {
           attr-node nan-length1 = INTEGER
           attr-node nan-length2 = 0
           attr-node nan-type = 'c'
           }
         | FIXED INTEGER
           {
           attr-node nan-length1 = INTEGER
           attr-node nan-length2 = 0
           attr-node nan-type = 'i'
           }
         | FIXED INTEGER
           {
           attr-node nan-length1 = INTEGER
           }
           INTEGER
           {
           attr-node nan-length2 = INTEGER
           attr-node nan-type = 'f'
           }
          ;
```

```
insert-clause: INSERTION IS AUTOMATIC
               {
               set-node nsn-insert = 'a'
               }
             | INSERTION IS MANUAL
               {
               set-node nsn-insert = 'm'
               }
              ;


retention-clause: RETENTION IS FIXED
                  {
                  set-node nsn-retent = 'f'
                  }
                | RETENTION IS MANDATORY
                  {
                  set-node nsn-retent = 'm'
                  }
                | RETENTION IS OPTIONAL
                  {
                  set-node nsn-retent = 'o'
                  }
                 ;


set-select-clause: empty
                   {
                   set-node nsn-select = 'o'
                   }
                 | SEMI-COLON SET SELECTION IS BY  set-select-spec
                  ;
```

```
set-select-spec: VALUE OF item-name IN record-type
                 {
                 if(valid-attr(item-name,record-type))
                   copy 'v' to set-select node select-mode
                   copy item-name to set-select node item-name
                   copy record-type to set-select node record1
                   copy BLANK to set-select node record2
                 end-if
                 else
                   print("Error-'item-name' not valid for 'record-type'")
                   perform yyerror()
                   return
                 end-else
                 }
               | STRUCTURAL item-name IN record-type
                 {
                 if(valid-attr(item-name,record-type))
                   copy 's' to set-select node select-mode
                   copy item-name to set-select node item-name
                   copy record-type to set-select node record1
                 end-if
                 else
                   print("Error-'item-name' not valid for 'record-type'")
                   perform yyerror()
                   return
                 }
                 EQ item-name IN record-type
                 {
                 if(previous item-name equals this item-name)
                   if(valid-attr(item-name,record-type))
                     copy record-type to set-select node record2
                   end-if
                   else
                     print("Error-'item-name' is not valid for 'record-type'")
                     perform yyerror()
                     return
                 end-if
                 else
                   print("Error-'item-name' items do not match")
                   perform yyerror()
                   return
                 end-else
                 }
               | APPLICATION
                 {
                 copy 'a' to set-select node select-mode
                 copy BLANK to record1, record2, item-name
                 }
                 ;
```

```
dml-statement: set-flag
             | move
             | get
             | find
             | store
             | connect
             | disconnect
             | erase
             | modify
             | perform-loop  /* not designed */
             | if-then      /* not designed */
               ;


set-flag: MOVE f-value TO f-name
        ;


f-value: YES
         {
         sys-flag-value = TRUE
         }
       | NO
         {
         sys-flag-value = FALSE
         }
        ;


f-name: EOF
        {
        eof = sys-flag-value
        }
      | NOTFOUND
        {
        notfound = sys-flag-value
        }
       ;
```

```
/* The MOVE statement is a COBOL assignment statement that assigns a */
/* value to a particular data field in a record template. We use a   */
/* list structure for this purpose.                                  */

move: MOVE item-value
      {
      if (first-move = TRUE)
        alloc and init move-list
        first-move = FALSE
      end-if
      create new data-item-node
      copy 'item-value' to value field in data-item-node
      establish curr-temp-item pointer
      }
     TO item-name
      {
      copy 'item-name' to name field in data-item-node
      }
     IN record-type
      {
      if (item-name not in record-type for current schema)
        perform error(2)
        return
      end-if
      else if ('record-type' node on move-list)
            connect curr-temp-item to record-template node
          end-else-if
          else
            create new record-template node
            copy 'record-type' to name field of record-template node
            connect curr-temp-item to record-template node
            add record-template node to move-list
            update curr-temp-rec pointer
          end-else
     ;

/* The GET statement takes the entire current record of the run unit */
/* or specified data fields of the current record of the run unit    */
/* and returns the values to the user.                               */

get: GET
      {
      alloc and init new 'get' node
      select-list = NULL  /* reset select-list */
      }
     mm
      ;
```

```
mm: item-list IN record-type
    {
    if ('record-type' is not equal to CIT.RUN-UNIT.type)
      perform error(3)
      return
    end-if
    else
      get-type = ITEMS in get node
      copy record-type to get node
      for (each data-item on item-list)
        if ('data-item' is not defined for record-type)
          perform error(2)
          return
        end-if
        else /* create pseudo tgt-list */
          copy data-item to get node
        end-else
      end-for
    end-else
    }
 | record-type
    {
    if ('record-type' is not equal to CIT.RUN-UNIT.type)
      perform error(3)
      return
    end-if
    else
      get-type = RETURN-ALL in get node
      copy 'record-type' to get node
    end-else
    }
 | empty
    {
    get-type = RETURN-ALL in get node
    copy CIT.RUN-UNIT.type to get node
    }
    ;

/* The FIND statements establish the current of run unit, record type, */
/* and set type.                                                       */

find: FIND  record-selection-expr  curr-suppression
    ;
```

```
/* The FIND ANY means: find any record of type record-type whose */
/* values for item1 through itemn match those in that record's  */
/* template in the user work area.                              */

record-selection-expr: ANY record-type
                       {
                       if ('record-type' record-template node is not
                          on move-list)
                         perform error(1)
                         return
                       else
                         alloc and init new 'find' node
                         find-type = ANY in find node
                         copy record-type to find node
                         alloc and init new abdl-str
                         alloc and init new tgt-list
                         /* begin forming a RETRIEVE request */
                         copy "[ RETRIEVE ((TEMPLATE = 'record-type')"
                           to abdl-str
                       end-if
                       select-list = NULL
                       }
                      USING  item-list  IN  record-type
                       {
                       if ('record-type' is same as previous 'record-type')
                         if (any data item on select-list is not
                           defined for record-type)
                           perform error(2)
                           return
                         end-if
                         else
                           create tgt-list item for all attributes
                             of 'record-type' record
                           for (each data item on select-list)
                             if ('data-item' not on move-list)
                               perform error(1)
                               return
                             end-if
                               else
                                 get 'item-value' from move-list
                                 concat "and ('data-item' = 'item-value')"
                                   to abdl-str
                               end-else
                           end-for
                           concat ")('tgt-list') by DBKEY]" to abdl-str
                           connect abdl-str to find node
                         end-else
                       end-if
                       else
                         perform error(6)
                         return
                       end-else
                       }
```

```
/* The FIND CURRENT means: Make the current of set-type the current */
/* record of the run unit.                                          */

              | CURRENT record-type WITHIN set-type
                {
                if (CIT.set-type.TYPE is not equal to 'record-type')
                  perform error(7)
                  return
                end-if
                else
               /* current of run-unit becomes current of 'set-type' */
                  alloc and init new 'find' node
                  find-type = CURRENT in find node
                  copy record-type to find node
                  copy set-type to find node
                  copy CIT.set-type.dbkey to find node
                end-else
                }

/* The FIND DUPLICATE means: Find the first record in the current set- */
/* type occurrence whose value for item1 through itemn matches those  */
/* for the same items in the current set-type occurrence, not the UWA */
/* record template. This implementation assumes the records being re- */
/* quested are already in a buffer.                                   */

              | DUPLICATE WITHIN set-type
                {
                alloc and init new 'find' node
                find-type = DUPLICATE in find node
                copy set-type to find node
                select-list = NULL  /* reset select-list */
                }
               USING item-list IN record-type
                {
                if ((record-type is not CIT.set-type.TYPE) or
                    (record-type is not CIT.set-type.member))
                  perform error(8)
                  return
                end-if
                else
                  copy record-type to find node
                  for (each data-item on select-list)
                    if (any data-item on select-list is not
                      defined for record-type)
                      perform error(2)
                      return
                    end-if
                    else /* create a pseudo tgt-list */
                      copy data-item to find node
                    end-else
                  end-for
                end-else
                }
```

```
/* This statement means: Find the FIRST, LAST, NEXT, or PRIOR record- */
/* type record within the current set-type occurrence. The ll token   */
/* takes the value FIRST, LAST, NEXT, or PRIOR.                       */
              | ll record-type WITHIN set-type
                {
                if ('record-type' is not a valid member type
                    for 'set-type')
                 perform error(5)
                 return
                end-if
                else
                 copy record-type to find node
                 copy set-type to find node

               /* RETRIEVE all member records of set occurrence */

                 alloc and init new abdl-str
                 alloc and init new tgt-list
                 copy "[RETRIEVE (
                      (TEMPLATE = CIT.set-type.member) and
                      (MEMBER.set-type = CIT.set-type.owner-dbkey))"
                      to abdl-str
                 create tgt-list for all attributes of member record
                 concat "('tgt-list') by DBKEY]" to abdl-str
                 connect abdl-str to find node
                end-else
                }

/* The FIND OWNER means: Find the owner of the current set-type occurrence */

              | OWNER WITHIN set-type
                {
                alloc and init 'find' node
                find-type = OWNER in find node
                copy set-type to find node
                alloc and init new abdl-str
                alloc and init new tgt-list

               /* form RETRIEVE request */

                copy "[RETRIEVE ((TEMPLATE = CIT.set-type.owner)
                     and (DBKEY = CIT.set-type.owner-dbkey))"
                     to abdl-str
                create tgt-list for all attributes of owner record
                concat "('tgt-list')]" to abdl-str
                connect abdl-str to find node
                }
```

```
/* This statement means: Find the first record-type record within the */
/* current set-type occurrence whose values for item1 through itemn   */
/* match the values found in the record-type template in the UWA, not */
/* the values in the current of set-type as in the FIND DUPLICATE.    */

        | record-type WITHIN set-type CURRENT
          {
          if ('record-type' not a member type of 'set-type')
            perform error(5)
            return
          end-if
          else
            alloc and init new 'find' node
            find-type = WITHIN in find-node
            copy record-type to find node
            copy set-type to find node
            alloc and init new abdl-str
            alloc and init new tgt-list

          /* begin forming RETRIEVE request */

            copy "[RETRIEVE ((TEMPLATE = 'record-type') and
                 (MEMBER.set-type = CIT.set-type.owner-dbkey)"
                 to abdl-str
            create tgt-list for all attributes of 'record-type'
              record
            select-list = NULL   /* reset select-list */
          end-else
          }
        USING item-list IN record-type
          {
          if (any data-item on select-list is not defined
              for 'record-type')
            perform error(2)
            return
          end-if
          else if (any data-item on select-list
                   not on move-list)
                 perform error(1)
                 return
          end-else-if
          else
            for (each data-item on select-list)
              get 'item-value' from move-list
              concat "and ('data-item' = 'item-value')
                to abdl-str
            end-for
            concat ")('tgt-list') by DBKEY]" to abdl-str
            connect abdl-str to find node
          end-else
          }
        ;
```

```
ll: FIRST
      {
      alloc and init new 'find' node
      find-type = FIRST in find node
      }
    LAST
      {
      alloc and init new 'find' node
      find-type = LAST in find node
      }
    NEXT
      {
      alloc and init new 'find' node
      find-type = NEXT in find node
      }
    PRIOR
      {
      alloc and init new 'find' node
      find-type = PRIOR in find node
      }
    ;


curr-suppression: LSQUARE supp-expr RSQUARE
            | empty
              ;


supp-expr: SUPPRESS UPDATE
       | UPDATE type-spec
         ;


type-spec: set-type
           {
           add set-type to suppression-list
           }
         | type-spec COMMA set-type
           {
           add successive set-types to suppression-list
           }
         ;
```

```
/* This statement means: Delete the current record of the run unit,    */
/* and all of its descendents regardless of whether they are owners of */
/* other sets.                                                         */

erase: ERASE ALL record-type
       {
       if ('record-type' is not CIT.RUN-UNIT.type)
         perform error(3)
         return
       end-if
       else
         alloc and init new 'erase' node
         erase-type = ALL in erase node
         for (each set-type in schema)
           if (CIT.set-type.owner-dbkey = CIT.RUN-UNIT.dbkey)
             member-type = CIT.set-type.member

             /* form RETRIEVE to get member records */
             alloc and init new abdl-str
             copy"[RETRIEVE(MEMBER.set-type = CIT.RUN-UNIT.dbkey)
                 (DBKEY) by DBKEY]" to abdl-str
             connect abdl-str to erase node

             /* erase member records */
             alloc and init new abdl-str
             copy"[DELETE((TEMPLATE = 'member-type') and
                 (DBKEY = ***))]" to abdl-str
             connect abdl-str to erase node

             /* delete all descendants of member records */
             perform erase-all(member-type,erase node)
           end-if
         end-for

         /* delete current of RUN-UNIT */
         alloc and init new abdl-str
         copy "[DELETE((TEMPLATE = 'record-type') and
              (DBKEY = CIT.RUN-UNIT.dbkey))]" to abdl-str
         connect abdl-str to erase node
       end-else
       }
```

```
/* This statement means: Delete the current record of the run unit if */
/* and only if, it is not the owner of a non-empty set.              */

   | ERASE record-type
      {
      if ('record-type' is not CIT.RUN-UNIT.type)
        perform error(3)
        return
      end-if
      else

        /* erase one record - current of RUN-UNIT */
        alloc and init new 'erase' node
        erase-type = NULL in erase node

        /* form RETRIEVE to see if 'record-type' is */
        /* owner of non-empty set                   */
        alloc and init new abdl-str
        copy "[RETREIVE(" to abdl-str
        first-time = TRUE
        for (each set-type in schema)
          if ('record-type' is owner type of set-type)
            if (first-time)
              concat "(MEMBER.set-type = CIT.RUN-UNIT.dbkey)"
                to abdl-str
              first-time = FALSE
            end-if
            else
              concat "or (MEMBER.set-type = CIT.RUN-UNIT.dbkey)"
                to abdl-str
            end-else
          end-if
        end-for
        concat ")(DBKEY) by DBKEY]" to abdl-str
        connect abdl-str to erase node

        /* for DELETE request */
        alloc and init new abdl-str
        copy "[DELETE ((TEMPLATE = CIT.RUN-UNIT.type) and
             (DBKEY = CIT.RUN-UNIT.dbkey))]" to abdl-str
        connect abdl-str to erase node
      end-else
      }
    ;
```

```
/* The STORE means: Create a new record in the database using values */
/* supplied by the user via MOVE statements, for the data items of   */
/* the specified record-type. The is connected to all sets in which */
/* INSERTION IS AUTOMATIC. The appropriate occurrence of the sets   */
/* must be selected before the new record can be connected. This is */
/* done based on the SET SELECTION clause specified in the database  */
/* schema definition for the sets in question.                       */

store: STORE record-type
       {
       if ('record-ype' record template node is not on move-list)
         perform error(1)
         return
       end-if
       alloc and init new 'store' node
       alloc and init new abdl-str
       copy "[RETRIEVE (" to abdl-str
       first-time = TRUE
       for (each data-item in schema for 'record-type')
         if (nan-dup-flag is set)
           if (data-item in move-list 'record-type' record template)
             get data-item value from move-list
             if (first-time = TRUE)
               concat"((TEMPLATE = 'record-type') and
                     ('data-item' = 'item-value'))" to abdl-str
               first-time = FALSE
             end-if
             else
               concat "or ((TEMPLATE = 'record-type') and
                       ('data-item' = 'item-value'))" to abdl-str
             end-else
           end-if
         end-if
       end-for
       concat")(DBKEY) by DBKEY]" to abdl-str
       connect retrieve request to store node
       alloc and init new abdl-str

             /* Form an INSERT request */
       copy"[INSERT (<TEMPLATE,'record-type'>,<DBKEY,***>" to abdl-str
       for (each 'data-item' in schema for 'record-type')
         if ('data-item' not on move-list for 'record-type')
           perform error(4)
           return
         end-if
         else
           get data-item value from move-list
           concat",<'item-name','item-value'>" to abdl-str
         end-else
       end-for
```

```
/* Now determine which set occurrences the new record belongs to  */
/* and add proper attribute-value pairs to the INSERT abdl-str to */
/* indicate set membership. The following FOR loop and CASE state-*/
/* ment fill the INSERT abdl-str with the proper pairs.           */
 for (each set-type in schema in which 'record-type' is a member)
  case (set selection mode) of

     /* set selection is by applicaton */
   a: perform by-application(INSERT abdl-str)

     /* set selection is by value */
   v: perform by-value(INSERT abdl-str)

     /* set selection is by structural */
   s: perform by-structural(INSERT abdl-str)

     /* no set selection criteria was given */
   o: perform by-default(INSERT abdl-str)

  end-case
end-for
concat "]" to INSERT abdl-str
connect INSERT abdl-str to store node
alloc and init suppression-list
}
curr-suppression
{
connect suppression-list to store node
}
;
```

```
/* The MODIFY means: Modify the entire current record of the run unit */
/* or the specified data items in that record.  The new values are    */
/* supplied by the user via the UWA.                                  */

modify: MODIFY
        {
        select-list = NULL   /* reset select-list */
        }
       item-list IN record-type
        {
        if ('record-type' is not current of run unit)
          perform error(3)
          return
        end-if
        if ('record-type' record-template node is not on move-list)
          perform error(1)
          return
        end-if
        if (any data item on select-list not defined for 'record-type')
          perform error(2)
          return
        end-if
        else
          alloc and init new 'modify' node
          locate record-template node on move-list for 'record-type'
          for (each data-item on select-list)
            alloc and init new abdl-str
            /* form UPDATE request */
            copy "[ UPDATE ((TEMPLATE = 'record-type') and
                 (DBKEY = CIT.RUN-UNIT.dbkey))" to abdl-str
            get 'item-value' from move-list
            concat "('data-item' = 'item-value')]" to abdl-str
            connect abdl-str to 'modify' node
          end-for
        end-else
        }
```

```
| MODIFY record-type
  {
  select-list = NULL  /* reset select-list */
  if ('record-type' not current of run unit)
    perform error(3)
    return
  end-if
  if ('record-type' record-template node is not on move-list)
    perform error(1)
    return
  end-if
  else
    alloc and init new 'modify' node
    for (each data-item in record-type)
      if (data-item not on move-list for 'record-type')
        perform yyerror(4)
        return
      end-if
      else
        alloc and init new abdl-str

        /* form an UPDATE request */
        copy "[ UPDATE ((TEMPLATE = 'record-type') and
          (DBKEY = CIT.RUN-UNIT.dbkey)) to abdl-str
        get new 'item-value' from move-list
        concat "('data-item' = 'item-value')]" to abdl-str
        connect abdl-str to 'modify' node
      end-else
    end-for
  end-else
  }
;
```

```
/* The CONNECT means: Connect the current record of the run unit to the */
/* current occurrence of the specified set type.  There may be several  */
/* sets listed in the statement.                                        */

connect: CONNECT record-type TO
        {
        if ('record-type' is not current of run unit)
          perform error(3)
          return
        end-if
        else
          alloc and init connect-list
        end-else
        }
        set-type-list
        {
        alloc and init 'connect' node
        for (each 'set-type' on connect-list)
          if ('record-type' is not a member type record for 'set-type')
            or (INSERTION is not manual)
            perform error(5)
            return
          end-if
          else
            alloc and init new abdl-str
            copy "[UPDATE ((TEMPLATE = 'record-type') and
                 (DBKEY = CIT.RUN-UNIT.dbkey))
                 (MEMBER.set-type = CIT.set-type.owner-dbkey)]"
                 to abdl-str
            connect new abdl-str to 'connect' node
          end-else
        end-for
        connect-list = NULL    /* reset connect-list */
        }
      ;


set-type-list: set-type
             {
             add 'set-type' to connect-list
             }
         | set-type-list  COMMA  set-type
             {
             add successive 'set-type'(s) to connect-list
             }
           ;
```

```
/* The DISCONNECT means: Disconnect the current record of the run unit */
/* from the set type occurrence that contains the record.              */

disconnect: DISCONNECT record-type FROM
          {
          if ('record-type' record is not current record of run unit)
            perform error(3)
            return
          end-if
          else
            alloc and init new connect-list
          end-else
          }
          set-type-list
          {
          alloc and init 'disconnect' node
          for (each set-type on connect-list)
            if ('record-type' is not a member type record for 'set-type')
              perform error(5)
              return
            end-if
            else
              alloc and init new abdl-str
              copy "[UPDATE ((TEMPLATE = 'record-type') and
                   (DBKEY = CIT.RUN-UNIT.dbkey))
                   (MEMBER.set-type = NULL)]"
                   to abdl-str
              connect abdl-str to 'disconnect' node
            end-else
          end-for
          connect-list = NULL  /* reset connect-list */
          }
        ;


perform-loop: PERFORM UNTIL bb EQ cc
        | END-PERFORM
          ;


bb: EOF
  | NOTFOUND
    ;


cc: YES
  | NO
    ;
```

```
item-list: item-name
           {
           put item-name on select-list
           }
         | item-list  COMMA  item-name
           {
           put successive item names on select-list
           }
         ;


schema-name: IDENTIFIER
           ;


record-type: IDENTIFIER
           ;


set-type: IDENTIFIER
         ;


item-name: IDENTIFIER
           ;


item-value: IDENTIFIER
         | INTEGER
           ;
```

```
proc error(err-code)
 /* This procedure prints error messages, causes data structures to */
 /* be de-allocated, and causes proc yyerror to be executed.       */


 case err-code of
  1: print("Error - must initialize 'record-type' record-template")

  2: print("Error - 'data-item' not defined in 'record-type'")

  3: print("Error - 'record-type' is not current record of run unit")

  4: print("Error - attempting to modify or store record without
          giving value of 'data-item'")


  5: print("Error - 'record-type' record does not belong to 'set-type'")

  6: print("Error - record-types specified are not the same")

  7: print("Error - 'record-type' is not current of 'set-type'")

  8: print("Error - 'record-type' must be a member record of 'set-type'")

 end-case
 perform cleanup()   /* free data structures */
 perform yyerror()
 return
end-error;
```

```
proc by-application(abdl-str)

  if (set-node nsn-insert ='a' )  /* insertion mode is automatic */
    concat",<MEMBER.set-type,CIT.set-type.owner-dbkey>" to abdl-str
  end-if
  else /* insertion mode is manual */
    concat",<MEMBER.set-type,NULL>" to INSERT abdl-str
  end-else

end-by-application;



proc by-value(abdl-str)

  locate data-item node in schema for set-select node item-name
    in set-select node record1
  if (nan-dup-flag set)
    get owner record type of set-type from schema
    if (owner record type node not on move-list) or
       (data-item not on move-list)
      perform error(4)
      return
    end-if
    else
      if (set node nsn-insert = 'a')  /* automatic insertion */
        get data-item value from move-list
        copy"[RETRIEVE((TEMPLATE = owner-record-type) and
            (item-name = 'item-value')) (DBKEY)]" to abdl-str
        connect new retrieve request to store node
        concat",<MEMBER.set-type,***>" to INSERT abdl-str
      end-if
      else  /* manual insertion */
        concat",<MEMBER.set-type,NULL>" to INSERT abdl-str
      end-else
    end-else
  end-if /* nan-dup-flag */

end-by-value;
```

```
proc by-structural(abdl-str)

  locate data-item nose in schema for set-select node item-name
    in set-select node record1 record-type
  if (nan-dup-flag set)
    get record1 name from set-select node for set-type
    if ('record1' record template node not on move-list) or
       (data-item not on move-list)
      perform error(4)
      return
    end-if
    else
      if (set-node nsn-insert = 'a')  /* automatic insertion */
        get data-item value from move-list
        get record2 name from set-select node for set-type
        copy"[RETRIEVE ((TEMPLATE = record2 name) and
            (item-name = item-value)) (DBKEY)" to abdl-str
        connect new retrieve request to store node
        concat",<MEMBER.set-type,***>" to INSERT abdl-str
      end-if
      else  /* manual insertion */
        concat",<MEMBER.set-type,NULL>" to INSERT abdl-str
      end-else
    end-else
  end-if  /* nan-dup-flag */

end-by-structural;



proc by-default(abdl-str)

  print("Warning - Attempting to store a record with no
        particular set selection given.  Will assume 'BY
        APPLICATION'")
  if (set-node nsn-insert = 'a')  /* automatic insertion */
    concat",<MEMBER.set-type,CIT.set-type.owner-dbkey>"
      to INSERT abdl-str
  end-if
  else  /* manual insertion */
    concat",<MEMBER.set-type,NULL>" to INSERT abdl-str
  end-else

end-by-default;
```

```
proc erase-all(record-type,erase node)

string   member-type;

  for (each set-type in schema)
    if ('record-type' is owner type of set-type)
      member-type = member type of set-type
      /* for RETRIEVE to get members of 'set-type' */
      alloc and init new abdl-str
      copy "[RETRIEVE(MEMBER.set-type = ***)(DBKEY) by DBKEY]"
        to abdl-str
      connect abdl-str to erase node
      /* delete member records */
      alloc and init new abdl-str
      copy "[DELETE((TEMPLATE = 'member-type') and (DBKEY = ***))]"
        to abdl-str
      connect abdl-str to erase node
      /* erase descendants of member records */
      erase-all(member-type,erase node)
    end-if
  end-for
  return(erase node)

end-erase-all
```

LIST OF REFERENCES

1. Demurjian, S.A. and Hsiao, D.K., "New Directions in Database- Systems Research and Development", in the **Proceedings of the New Directions in Computing Conference**, Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval postgraduate School, Monterey, California, February 1985.

2. Hsiao, D.K., and Menon, M.J., **Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part I)**, Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

3. Hsiao, D.K., and Menon, M.J., **Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part II)**, Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

4. Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines", **Proceedings of National ACM Conference** , 1978.

5. Benson, T. P. and Wentz, G. L., **The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System** ,M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

6. Kloepping, G. R. and Mack, J. F., **The Design and Implementation of a Relational Interface for the Multi-Lingual Database System** ,M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

7. Date, C.J., **An Introduction to Database Systems** , pp. 389-446, Addison Wesley, 1982.

8. Tsichritzis, D. C. and Lochovsky, F. H., **Data Models**, pp. 119-147, Prentice-Hall, 1982.

9. Olle, T.W., **The CODASYL Approach to Data Base Management** , John Wiley & Sons, Inc., 1978.

10. Hsiao, D.K., and Haray, F., "A Formal System for Information Retrieval from Files", **Communications of the ACM** , Vol. 13, No. 2, February 1970, Corrigenda, Vol. 13, No. 3, March 1970.

11. Banerjee, J., Hsiao, D. K., and Kerr, D. S., **DBC Software Requirements for Supporting Network** Databases,

Technical Report OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, November 1977.

12. Data Base Task Group of CODASYL Programming Language Committee Report (April 1971).

13. Meyer, G. and MacDougal, P., **An Attribute-value Translation of CODASYL's Data Manipulation Language** , Ohio State University, 1982.

14. Johnson, S. C., **YACC: Yet-Another-Compiler Compiler** , Bell Laboratories, Murray Hill, New Jersey, July 1978.

15. Lesk, M. E. and Schmidt, E., **Lex - A Lexical Analyzer Generator** , Bell Laboratories, Murray Hill, New Jersey, July 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7.	Clemon R. Wortherly 28 "L" Street Sumter, South Carolina 29150	3